JOEL ON SOFTWARE

JOEL说软件

[美]Joel Spolsky 著
谭明金 王平 译

J O E L   O N   S O F T W A R E

電子工業出版社

目錄

Joel 说软件

整理自Joel on Software Translation Project。

这是一本介绍软件管理的小品文集。全书分为45章，每章就是一个独立的专题或者知识点。本书内容十分丰富全面，小到项目负责人制订进度表，大到软件执行总裁提出富有竞争性的战略，都在本书的介绍之列。尽管内容很多，但事例驱动的写作方式，奠定了本书在可学性与可用性方面明显的优势。本书从不同侧面满足了软件开发人员、设计人员、管理人员及从事软件相关工作的人员的学习与工作需要。

第一部分 位与字节：编程实践点滴

一、语言的选择

为什么开发人员要针对给定任务选取某种编程语言，而不去选择其他的语言？

当项目需要抢速度时，我往往选用原始的C语言进行开发。

如果开发要求运行在Windows上的、版本尽可能小的程序时，一般选用带有静态连接MFC 的C++语言进行编程。

假如需要开发在Mac、Windows与Linux等*作系统上运行的图形化界面（GUI)程序时， Java语言就成为通常的选择。这样得到的GUI虽然算不上最好，但是它能够工作。

对于需要快速开发出来的GUI和确实显得自然的UI来说，可以选用Visual Basic作为程 序设计语言，但这样做，要清楚地认识到分发大尺寸的程序所付出的代价，以及要受制于 Windows 的事实。

假使需要开发运行在任何UNIX机器上而速度不必很快的命令行程序，那么用Perl语言进 行开发是一个不错的选择。

若程序必须在浏览器内运行，那么JavaScript就变成惟一选择。在SQL存储过程中，通 常需要选取某提供商所专有的SQL派生功能，或者说，干脆回家走人。

关键点是什么？

不过，我几乎从来都不会基于语法而选取某种语言。是的，我比较偏爱提供了{};要素 的那些语言（C/C++/C#/」ava)。并且，在关于什么是“好的”语法方面，我是有些看法的。 但是，我不会认可仅仅因为要使用分号的缘故而增加一个20MB大小的运行库的做法。

是什么使我对.NET的跨语言策略感到有点惊讶的呢？是这样一种理念：可以选用任何语 言进行软件开发，语言的种类可以是亿万，并且它们的工作方式都相同。

VB.NET与C#.NET除了语法上的细微差异之外，实际上是一样的。其他语言要想成为.NET 世界的一部分，就得至少需要支持一个核心功能和类型集，否则它们将不能与其他语言很好 地链接。问题是，如何开发.NET中的UNIX命令行应用程序？如何在.NET中开发小于16 KB的Windows微程序呢？

看起来，在.NET中选择语言，我们似乎就不得不要关注一下语法了。

二、深入底层

我在自己的Web站点上花了许多时间聊关于令人激动的“巨幅图画”素材方面的内容， 比如说.NET与Java啦，XML策略啦，锁定啦，竞争策略啦，软件设计啦，体系结构啦，不一 而足。所有这些素材在某种意义上讲，都是一块夹心蛋糕。位于顶层的是软件策略，接下来 是诸如.NET之类的体系结构，然后是各种产品，即Java之类的软件开发产品，或者Windows 之类的平台。

请继续往蛋糕的底层看。是动态连接库DLL?对象？函数？不！还要看得更低一些！在 某个点上，要考虑用编程语言书写的代码行。

这样的层次还不够低，今天我要考察的是CPU—一不停地向四周传输字节的小块硅片。 假设您是一位编程新手，现在不妨抛弃业已建立的编程、软件与管理等方面的所有知识，重 新回到底层的冯?诺依曼（Von Neumann)基本素材上来，并且暂时将」2EE踢出脑海，而只 考虑“字节”。

这样做的意义何在？我觉得，人们所犯的一些最大错误（即使在体系结构的最高层）的 根源在于，对处于最底层的几个简单事物理解不够或者一知半解。虽然金碧辉煌的宫殿已经 建起来了，但地基却是一团糟。所用的建材不是上好的水泥板，而是零乱的碎石。于是乎， 宫殿虽然很好看，但浴缸不时在浴室的地面滑来滑去，而让人不知所措。

好了，现在做一次深呼吸。看着我，这个小小的练习将用C语言来完成。

记住C语言中字符串的工作方式：字符串由系列字节组成，后跟一个取值为0[1]的空 (null)字符。这里很明确地表明了两点：

1. 如果不遍历字符串并查找末尾的空字符，就没有办法知道字符串在何处结束（即字 符串长度）。
2. 字符串中不能包含任何零。因此，C字符串中不能存放诸如JPEG图片之类的任何二进制数据块。

为什么C字符串要以这种方式工作？那是因为在上面开发了 UNIX与C程序设计语言的 PDP-7微处理器有一种ASCIZ字符串类型。ASCIZ的中文意思是“末尾有一个零的ASCII”。

这是存放字符串的惟一途径吗？不是。事实上，这是存放字符串最差的方式之一。对于 重要程序、API、操作系统与类库，用户应该像躲避瘟疫一样地避开使用ASCIZ字符串。为 什么呢？

下面从编写函数strcat的一个代码版本入手进行讨论。该函数的功能是将一个字符串附 加在另一个字符串之后。

```
void strcat( char* dest, char* src )
{
while (*dest) dest++;
while (*dest++ = *src++);
}
```

对代码稍作研究，就可以明白这个函数正在做些什么。首先，它遍历第一个字符串以寻 找空终止字符。一旦找到空字符，就逐字符将第二个字符串一次性拷贝到第一个字符串之后。

虽然这种*作与串接字符串的方法对于Kernighan与Ritchie[2]来说已经足够好了，但 是也有它的问题。比如，假设有一串名字要在一个大字符串中串接起来，那就会暴露出来一 些问题：

```
char bigString[1000]; /*永远也不知道需要分配多大的存储空间...*/
bigString[0] = '\0';
strcat(bigString,MJohn,");
strcat(bigString,"aul,");
strcat(bigString,"George,");
strcat(bigString,"Joel ");
```

这能够应付，不是吗？是的。并且，看起来也很漂亮整洁。

函数性能如何？能够运行得尽可能快吗？可扩展性如何？要是有一百万个字符串要追 加，这样处理还算得上很好的方式吗？

不，该代码使用的是蹩脚的Shlemiel喷涂算法。Shlemiel是谁？他是下面这则笑话中的 人物：

Shlemiel得到一份当街道油漆匠的工作，工作内容是在马路中间喷涂点画线。第一天， 他拿出一罐漆来到他负责的路段，喷涂了 300码长的线。“干得不错！ ”他的老板称赞道， “真是一位麻利的工匠”，然后赏给他一个戈比（一种俄罗斯辅币，译者注）。

第二天，Shlemiel只喷涂了 150码。“喏，虽然不如昨天那样好，但你仍然算得上一位 麻利的工匠！ 150码还是值得肯定的一个长度，”老板说完又赏给他一戈比。

接下来的一天，Shlemiel只喷涂了30码长的马路。“才30码！”他的老板吼道。“这太 令人难以接受了！第一天你干的工作量是今天的10倍！接下来是怎么回事？〃

“我尽力了，”Shlemiel说道。“一天一天下去，我离油漆罐越来越远！〃

(对于规模特大的应用来说，到底要“赊”多大存储空间[3]？)这则蹩脚的笑话很贴 切地说明了，如果像前面给出的代码那样使用strcat函数将会出现什么结果。由于strcat 的第一部分代码必须每次都扫描整个目的字符串，以反复寻找那个捉摸不定的空终止字符， 因此该函数将比所希望的速度慢得多，并且它根本谈不上存在伸缩性。你每天使用的许多代 码都有这个问题。很多文件系统是以一种在一个目录中放太多的文件的不恰当方式来实现的。 说它不恰当的原因在于，一旦目录中达到成千上万个条目时，其运行性能就开始急剧下降。 试着打开一个装填过度的Windows垃圾箱来看看运行情况如何——要花几个小时才能显示出 文件，它与所包含的文件数目基本上不成线性关系。这里面的某个地方必定存在一个Shlemiel喷涂算法。不管

什么时候，只要某个应用项目看起来应该具有线性性能，可它却表 现出n2的性能，那么就得寻找隐藏于其中的那些Shlemiel们。它们通常是被库文件隐藏起来 了。看到一个循环体中出现一列或者一个strcat函数未必要大声嚷嚷“找到n2 了”，但那 一定是问题之所在。

如何修正strcat函数呢？有几个聪明的C语言程序员是这样来实现他们自己的mystrcat 函数的：

```
char* mystrcat( char* dest, char* src )
{
while (*dest) dest++; while (*dest++ = *src++); return --dest;
}
```

这里的代码做了些什么？以非常小的额外代价，就返回了一个指向新的较长字符串尾部 的指针。如此一来，调用该函数的代码就能够决定进一步处理字符串附加*作，而不用首 先重新扫描整个字符串了：

```
char bigString[1000]; /*永远不知道要分配多少存储空间*/
char *p = bigString;
bigString[0] = '\0';
p = mystrcat(p'John,");
p = mystrcat(p,"aul,");
p = mystrcat(p,"George,");
p = mystrcat(p,"Joel ");
```

这个性能当然是线性而不是n2的，因此它不会在串接许多字符串时而导致性能下降。

Pascal语言的设计人员意识到了这个问题，并通过将字符串的首字节用于存放字节个数，对该问题加以解决。由此得到的字符串称为Pascal字符串。这样的字符串可以包含取值为0 的字节，它不会被当做空终止字符。由于一个字节可以表示的最大数值为255,因此Pascal 字符串只能限于255个字节的长度。但由于它们不是以空字符作为终止符，所以占据与ASCIZ 字符串相同数量的存储器。Pascal字符串最主要的亮点在于用户完全没有必要仅仅为了弄清 字符串的长度而使用一个循环语句体。确定一个Pascal字符串的长度只用到一条汇编指令， 而不是整整一个循环体。这是非常快的。

旧的Macintosh*作系统处处用到了 Pascal字符串。许多C程序员在其他平台上也用到 了Pascal字符串以提高处理速度。Excel在内部使用Pascal字符串，这就解释了为什么Excel 中许多地方的字符串限于255个字节的原因，同时它也是Excel的速度快得出奇的一个原因。

在相当长的一段时间内，如果想在C代码中放一个Pascal字符串文本，就不得不写一条 如下形式的代码：

```
char* str = "\006Hello!";
```

对啰，用户得亲自动手计算字节的数目，并将它以编码的形式放在字符串的第一个字节 之中。偷懒的程序员会按如下形式来实现该功能，但得到的程序运行起来很慢：

```
char* str = "*Hello!";
str[0] = strlen(str) - 1;
```

需要引起注意的是，在这种情况下得到的字符串，同时也是以空字符作为终止符的（由 编译器来做这件事）。我习惯于将它们称为杂交串，因为这比称它们是以空字符结尾的Pascal 串要简洁一些。不过，这是一个高速变化的频道，因此将来得使用较长的名称。

前面忽略了一个重要的问题。还记得这样的一个代码行吗？

```
char bigString[1000]; /*从不知道要分配多少存储空间   */
```

既然现在谈到了位，就不应该忽略掉这个问题。我本来应该正确地处理该问题：弄清楚 到底需要多少个字节，并分配合适的内存量。

难道不应该吗？

大家知道，如果不这样做，那么聪明的黑客会读取该代码，并注意到用户只分配了 1000 个字节而觉得它够用了。黑客们还会找到一些比较聪明的方式，诱骗用户将1100个字节的字 符串用strcat放到1000个字节的内存中，从而重写堆栈页面并改变返回地址。结果造成在该 函数返回时，它执行了黑客自己写的一些代码。这就是他们在说某特殊程序具有缓冲区溢出 易感性（buffer overflow susceptibility)时所谈论的内容。在那一段Microsoft Outlook 使黑客攻击行为容易得让十来岁的小孩也可以做到之前的岁月里，这曾经是出现黑客和蠕虫 的首要成因。

好了，看来那些程序员都不过是些跛脚驴。他们本来应该去弄清需要分配多少内存的。

不过，说实在话，对于一般程序员而言，C语言并没有使该问题变得容易解决。还是回 过头来看看前面有关那个披头士的例子吧：

```
char bigString[1000]; /*从不知道要分配多少内存*/

char *p = bigString;
bigString[0] = '\0';
p = mystrcatCp'John,");
p = mystrcat(p,"aul,");
p = myst「cat(p,"Geo「ge,");
p = mystrcat(p,"Joel ");
```

应该分配多少内存？不妨试着以合理的方式来做这件事。

```
char* bigString; int i = 0;
i = strlen("John,")
+ strlen("aul,")
+ strlen("George,")
+ strlen("Joel ");
bigString = (char*) malloc (i + 1);
/*别忘了存放空终止字符也要空间！ */
```

我的眼睛花了。或许读者已经准备换频道了。我不责怪你，但烦请忍耐一下，因为真正 令人感兴趣的内容到来了。

仅仅为了弄清字符串有多大，就必须一次扫描完所有的字符串，然后在串接时还要再次 对它们进行扫描。如果使用Pascal字符串，至少strlen*作是很快的。也许，我们可以编 写一个能够为我们重新分配内存的strcat函数版本。

它将另外一整罐蠕虫（内存分配器）打开。你知道malloc函数是如何工作的吗？ malloc 函数的实质体现在，它有一个将可用的内存块连接为一个长长的列表的所谓空闲链表。调用malloc函数时，它沿连接表寻找一个大到足以满足用户请求所需要的内存块。然后，将该内存块一分为二（一块的大小与用户请求的大小相等，另一块的大小就是剩下的字节）。接下来，将分配给用户的那块内存传给用户，并将剩下的那块（如果有的话）返回到连接表上。调用free函数时，它将用户释放的内存块连接到空闲链上。到最后，空闲链会被切成很多的小内存片段，如果这时用户申请一个大的内存片段，那么空闲链上可能没有可以满足用户要求的片段了。于是，malloc函数请求延时，并开始在空闲链上翻箱倒柜地检查各内存片段，对它们进行整理，将相邻的小空闲块合并成较大的内存块。

这要用去三天的时间。经过一通“鸡飞狗跳”的混乱状况之后所得到的最终结果变成： malloc的性能表现为永远也快不起来（总是要遍历空闲链），有时候甚至显得不可预测，在 清理内存时慢得使人害怕。（顺便说一下，这与垃圾收集系统的性能特征类似，不令人吃惊 才怪呢。可见，人们做出的关于垃圾收集行为如何导致性能损失的所有断言都不完全成立。 因为典型的malloc实现形式具有同样的性能损失，尽管略显温和。）

聪明的程序员通过总是分配大小为2的幂的内存块，而最大限度地降低潜在的malloc性 能丧失。也就是说，所分配的内存块大小为4字节、8字节、16字节、18446744073709551616 字节，等等。至于说原因，任何与Lego打过交道的人应该是一目了然的，这样做最大限度地 减少了进入空闲链的怪异片段（各种尺寸的小片段都有）的数量。尽管看起来这好像浪费了

空间，但也容易看出浪费的空间永远不会超过50%。因此，程序不会使用大小超过所需尺寸两倍的内存，这并不是大得不得了。

现在，不妨假设自己要编写可以自动重新分配目的缓冲区的“智能〃st「cat函数。总是 应该分配所需要的实际尺寸吗？我的老师兼顾问Stan Eisenstat [4]建议，在调用realloc 函数时，应分配出两倍于以前所分配的内存。这意味着，用户决不会调用「ealloc函数达lg (n)次以上。该性能特征即使对于大型字符串也可以忍受，并且决不会浪费多于50%的内存。

总而言之，在字节领地里，越往下走，生活就显得越来越凌乱。你难道不对自己再也用 不着使用C语言编写内存分配函数而感到高兴吗？我们拥有Perl、Java、VB与XSLT这些伟大 的编程语言，它们再也不会让你去考虑此类事情。不用问为什么，它们只管去做好了。

不过，竖直的基础结构偶尔也会从起居室的中部凸现出来，于是我们就得考虑是否使用 String类，或者StringBuilde「类，或者相关方面的一些差别，因为编译器仍然没有聪明到 能够理解我们正在尽力完成的一切，从而试图帮助我们不在无意之中写出一些Shlemiel喷涂 算法。

这篇随笔文章在我写了一篇说明以XML形式存储的数据不能用SELECT author FROM books 实现性能很快的SQL语句的即兴网评[1]之后，受到了不友善攻击。正是因为大家没有理解我在说些什么，以及既然我们整天都在CPU周围打滚，这个主张可能才更显得有意义。

关系数据库是如何实现SELECT author FROM books的？在关系数据库里，关系表（例如books表）的每一行具有相同的字节长度，而每个字段距离各行开头的偏移量是大小固定的。因此，如果books表的每条记录是100个字节长，并且autho「字段处在偏移量为23的位置，那么作者们的名字就存放在第23, 123, 223, 323等位置的字节处。移到该查询结果的下一 条记录的代码是什么？大体上讲，这条代码为：

```
pointer += 100;
```

只用了一条CPU指令！快，快，实在是快！

现在我们来看XML中的books表。

```
<?xml blah blah>
<books>
<book>
<title>UI Design for Programmers</title> <author>Joel Spolsky</author>
</book>
<book>
<title>The Chop Suey Club</title> <author>Bruce Weber</author>
</book>
</books>
```

马上就想到的另外一个问题是，移到下一条记录的代码是什么？

唔……

在这一点上，好的程序员会说，嗒，将XML解析成内存中的树结构吧，以便能够相当快 速地处理它。在这里，针对SELECT author FROM books语句，CPU必须完成的工作量绝对会 将人折磨至疯。正如每个编译器设计人员都知道的那样，语法分析与解释是在编译处理过程 中最慢的部分。只要谈我们在解释、分析与建立抽象的内存语法树时，发现它涉及许多处理 起来很慢的字符串素材，以及许多执行起来很慢的内存分配内容就够了。

况且，这还假定了有足够的内存用于一次性加载整个内容。对于关系数据库来说，在记 录之间执行移动*作的性能是固定不变的，实际上它不过是一条指令的事。这是费了好大 的力气有意为之的。同时，因为得益于内存映像文件，用户只需加载将要实际使用的那些磁 盘页面。

对于XML来说，如果要做预分析的话，那么在记录之间执行移动操作的性能是固定不变 的，只是启动时间极大；如果不做预分析，那么在记录之间执行移动操作的性能根据它前 面的记录长度而变化，并且CPU指令长达好几百条。

这在我看来意味着，如果用户讲究性能，并且数据量很大，那么就不能使用XML。如果 只有一点儿数据，或者事情做得不必很快，那么XML不失为一个好的形式。并且，如果用户 确实希望鱼与熊掌兼得，就必须找出一种途径用于紧*XML存放元数据。元数据跟Pascal 字符串中用于计数的字节的作用差不多，它向用户给出提示信息，以便不用去分析与扫描字 符串就能确定它们在文件的什么地方。当然，这样一来用户就不能使用文本编辑器去编辑文 件了，因为那会搞乱元数据，从而它不再是真正的XML了。

对于听众中三个仍然在这一点上支持我的和颜悦色者，我希望你们已经学到了些东西或 者重新考虑过一些事情。我希望针对诸如strcat与malloc函数到底如何工作之类的那些令人 厌烦的一年级计算机课程素材所做出的思考，已经向你们提供了新的工具手段，从而在处理 XML之类的技术问题时，制定出最新的关于顶层策略和体系方面的决定。

作为家庭作业，思考一下为什么Transmeta芯片总是会觉得行动迟缓？为什么关于 TABLES的原始HTML说明设计得如此之差，以致Web页面的大型表不能够快速地显示给使用调 制解调器的人们？为什么COM是如此棒，但在跨越进程边界时却不是这样的？还有，为什么 NT人将显示器驱动程序放在内核空间，而不是用户空间？

所有这些事情都要求用户去思考字节，字节影响着用户在各种体系与策略方面做出决定。 这就是我为什么坚持一种教学观点——大学一年级学生需要从基础学起，即用C语言以及从CPU开始向上逐步构建自己程序设计技能——的原因。我真的从心底里厌烦在多得出奇的计算机课程计划中，将Java看做是一门好的入门语言的做法。这些计划的理由是：Java语言虽然很“简易”且不会陷入所有那些令人厌烦的字符串与malloc素材之中，但是可以学习特别 棒的OOP知识，从而使大型程序具有如此之多的模块化特性。

这是一场即将发生的教学灾难。一届又一届的毕业生正在侵袭我们，他们正在忽左忽右 地创建着Shlemiel油漆匠算法，而他们甚至还没有意识到这一点。原因在于，他们根本就没 有那种字符串在深层次上处理起来很困难的理念，即使在Perl脚本上也不能很好地看到那一 点。如果想在某个方面把别人教好，自己首先得从最底层开始研究。这就像空手道功夫小子 要做的事情一样。打蜡，剥蜡。打蜡，剥蜡。这样的过程要进行三个星期。然后，他就可以 轻松击败其他小孩了。

三、joel测试：改进代码的12个步骤

你曾经听说过SEMA吗？ [1]它是一个相当深奥的系统，用于测试软件团队优良程度。不，等一下！不要去读SEMA!要理解这个东西就得花大约6年的时间。因此，我自己提出了一套 不很严格的宽松测试来评估一个软件团队的质量。该测试的闪光之处在于它只需花费大约三 分钟的时间。将节省下来的所有时间利用起来，你就可以进入医学院了。

Joel测试

1. 使用源控制机制吗？
2. 能一步完成连编吗？
3. 每天都做连编吗？
4. 有故障信息数据库吗？
5. 在编写新代码之前修复故障吗？
6. 有最新的进度表吗？
7. 有规格说明书吗？
8. 程序员拥有安静的工作环境吗？
9. 你用到了你资金能力内可买到的最好工具吗?
10. 有测试人员吗？
11. 新聘人员在试用期写代码吗？
12. 进行走廊可用性测试吗？

Joel测试的巧妙之处在于，针对每个问题，都可以很容易得到“是”或者“不”的回答， 没必要弄清每天的代码行，或者每个测试点的平均故障数。对每个做出“是”的回答，给团 队加上一分。Joel测试的不足之处是，确实不应该用它来保证核动力工厂软件是安全的。

得12分是极好的，11分可以容忍，但10分或者更低就是严重的问题了。现实情况是，大 多数软件机构正运转在2或3分上，它们需要正式的帮助，因为像Microsoft这样的公司全 天候运转在12分上。

当然，这些内容不是决定成功或者失败的惟一因素。特别是，假设一个大型的软件团队正在努力开发一个没人想要的产品，对了，人们就是不会要它。这就好比是一支由“枪手”组成 的队伍，他们做的事情与此毫不相干，却仍然设法生产能够改变世界的令人难以置信的软件。但是，万事万物都有可比性，如果将这12件事情办妥了，那么就有了一个可以随时投入使 用的训练有素的团队。

1.使用源控制机制吗

我使用过商业版本的源控制包与免费的CVS[2]，并且，我要说CVS不错。但是，如果不使 用源控制，那就得十分强调尽力让程序员在一起工作。程序员没有办法知道其他人员做了什么，所以不容易进行错误回滚。源控制系统的另外一个巧妙之处在于，可保证源代码在每个

程序员的硬盘上都是经过自动确认了的——我从来没有听说使用源控制的项目丢失过许多 代码。

2.能一步完成连编吗

对于这一点，我的意思是：从最新的源快照进行一次可分发的连编，需要多少个步骤？ 一个优秀的团队仅有一个能够做从头至尾的完全校验的有效脚本；可重建每个代码行；按所有不同的版本、语言组合来建立EXE文件；创建安装包；生成最终介质——CD-ROM规 划；下载Web站点，等等。

如果这样的过程需要多步才能完成，那么就很可能出错，并且越接近产品分发时刻，就越希望修复“最后”故障、形成最终EXE文件等*作所经历的周期会更短一些。如果要用20 步才能完成代码的编译与安装程序的生成等任务，那么你因为感到烦躁而很可能犯低级错误。

正是因为这个原因，我供职的最后一家公司就从当初使用Wise到现在改用InstallShield 了 ：我们要求安装过程能够通过使用NT调度表通宵自动地从脚本运行，而Wise不能在晚上 从调度表启动，因此我们将它扔掉了。（友善的Wise人向我保证，他们的最新版本一定支 持夜晚连编功能。）

3.每天都做连编吗

使用源控制时，程序员时常不经意地加入一些东西，导致连编中断。比如说，他们添加了一个新的源文件，但是忘记了将该源文件添加到代码库中去。这在他们自己机器上编译是很好的，于是，他们锁上机器回家去了，心情舒畅而惬意。不过，别人却不能工作，但也只好回家去，心情郁闷而忧心忡忡。

中断连编过程是如此有害（同时也如此常见），以致它反而有助于保证每天的连编不至于出现中断没有被觉察的情况。在大型软件团队中，确保中断得以立即修复的一条很好的途径是， 在每天的午餐时间实施每天都要进行的连编过程。每个人在午餐之前都尽可能多地进行加注 新内容的工作。当他们回来时，兴许连编已经做完了。要是这个办法可行，就好极了！大家 在核实最新的源文件版本以后继续工作。如果连编失败了，就得进行修复，不过，也可以从 连编过的没有发生中断的源文件版本重新再来。

我给Excel开发团队制订了一条规则，无论谁中断了连编，都要负责担当连编*作的临时 保姆以示“惩戒”，直到有别人中断连编过程为止。这是一条激励大家不要中断连编过程的 好机制，也是让大家轮流进入连编进程的好途径，从而使每个人弄清连编是如何进行的。

关于每日连编方面的更多内容，可阅读我写的《每日连编是朋友》[3]—文。

4.有故障信息数据库吗

我并不在乎你说些什么。如果你正在开发代码（即使是只有一个人的团队），但是没有一个组织严密，并列举了代码中所有已知故障的数据库，那么分发出去的代码的质量可能是很低的。许多程序员认为能够做到在自己的头脑中放一份故障信息表。这是胡说！反正我一次记

不住两条或者三条以上的故障信息，也许就在第二天早上或者代码分发的一刹那，将它们忘得干干净净。可见，绝对需要正式地保存一份故障信息记录。故障信息数据库可以很复杂，也可以很简单。一个可用的故障信息数据库必须至少为每个故障包含如下数据：

- 重现故障的完整步骤
- 预期功能
- 观察到的（故障）行为 一要分配给谁 一是否已修复

如果故障跟踪软件的复杂性是阻止你跟踪故障的惟一因素，那么建立一个包含上述关键信息的五字段关系表，然后开始使用它。

关于故障跟踪方面的更多内容，可阅读《故障轻松跟踪》一文[4]。

5.在编写新代码之前修复故障吗

Microsoft Word的第一个真正Windows版本曾被认为是一个“死亡之旅”项目。它花费的 时间不计其数，并且常常脱开正常轨道。整个开发团队行进在看不到尽头的工作泥沼之中， 项目经过了延迟、延迟与再延迟，压力大得难以置信。几年以后，当这个响当当的物件终于 分发下去时，微软将整个开发团队送到迦南去度假，从而坐下来做某种严肃而深刻的灵魂自 我反省。

他们意识到，项目经理们一直是如此固执地坚持按时间表行事，以致使程序员只顾赶进度而编写出极其糟糕的代码，因为故障修复阶段并没有成为正式进度表的一部分。整个团队没有在缩减故障数量方面做出任何努力，而是与此恰恰相反。

整个情景演变为：一个必须通过写代码来计算文本行高度的程序员只是简单地给出”return 12 ”这样的代码了事，然后就等故障报告出来，故障报告总是说他的函数不正确。到头来， 进度表只不过成了一张功能故障列表。在尸检行业，这称之为无限缺陷法（infinite defects methodology)。 为纠正出现的问题，微软公司普遍米取了一种称之为零缺陷法(zero defects methodology) 的措施。公司的许多程序员对此报以咯咯一笑，因为这听起来就像是一种单纯依*行政命 令就能减少故障数目的管理思想。其实，零缺陷意味着在任意给定时间点，最需要优先去做 的事情是在写任何新的代码之前消除故障。下面说说这样做的理由。

一般说来，在修复故障之前等待的时间越长，付出的代价（时间与金钱上）就越大。

比如，当编译器捕捉到一个拼写或者语法错误，然后去修正它基本上是小事一粒。

在首次试图运行代码时所看到的错误，可以毫不费事地立即进行修正，因为所有代码在头脑中仍然十分清晰。

如果在几天前编写的代码中发现了问题，找到它确实需要花费一些工夫。不过，只要重新阅读编写的代码，就可以回想起所有的事情，从而在合理的时间里把它修正过来。

但是，假如在几个月以前写的代码中发现了问题，就可能已经忘记了代码方面的许多事情，要修复故障就困难多了。这个时候倒是可以去对别人的代码进行纠错，编写代码的人可能正在阿鲁巴岛（阿鲁巴岛是拉丁美洲荷属安的列斯群岛中的大岛，译者注）度假。在这种情况下，修理故障就跟从事科学工作一样：你得慢慢地、系统而小心翼翼地行事，并且不能确定要花多长的时间才能纠正错误。

然而，要是在已经分发出去的代码中找到了缺陷，那么要修复它所要付出的代价将是难以置信的。 这就是要立即修复故障的一个理由：因为这样做花费的时间较少。下一个理由与这样一个事 实有关：预测要花多长时间去编写新代码比预测要用多少时间去修复现有故障容易得多。

比如，假设叫你预测要花多长时间去编写一段进行列表分类排序的代码，你一定能够给出一个相当好的估计。但是，如果叫你预测要花多少时间去修复代码在安装了 Internet Explorer 5.5版本以后无法工作的故障，你恐怕连猜也猜不出来，因为你不知道（从概念 上讲）故障出现的原因。很可能要用三天的时间去把故障跟踪出来，也可能只要两分钟就够 了。

这里要表达的意思是，如果进度表包含了许多有待修复的故障，那么该进度表就是不可* 的。但是，如果你已经修复了所有已经知道的故障，并且剩下的就是编写新代码，那么进度 表就是极其准确的。

关于将故障数目维持在零的措施有另外一个重大意义，可以对竞争做出快得多的反应。一些程序员一直将这看做是让产品做好分发的准备。然后，如果竞争对手加入一个“杀手锏”新 功能来抢夺顾客，你就可以只去实现此新功能，并即时分发产品，而用不着去修复一大堆累 积起来的故障。

6.有最新的进度表吗

这个因素促使我们按进度表推进。如果代码对业务显得十足重要，那么必定有许多关于它为什么对业务重要的理由可用以了解代码要在何时完成。程序员在制订进度表方面所显现出来的执拗，给他们留下了坏名声。“在该做的时候自然就做了！”他们冲着业务人员大声喊叫道。

遗憾的是，事情远还没有完。在分发代码之前，需要业务部门去很好做出的规划决策太多了， 这包括产品演示、贸易展示与广告等。做这件事的惟一途径就是拥有一个进度表，并保证它 始终反映最新的情况。

拥有进度表的另外一个至关重要之处在于，它首先迫使你做出将要实现什么功能的决定，然后强迫你拣出最不重要的功能，并加以剪除而不是滑入功能沼泽地带（也就是功能范围蔓延开来）[5]。

持有进度表并不意味着必定显得困难重重。不妨阅读一下第9章，其中给出了一种创建大进 度表的简单方式。

7.有规格说明书吗

写规格说明书好比是理乱麻：人人都认为是好事情，可就是没人去做。

虽然我吃不准这是为什么，但觉得很可能是因为大多数程序员都讨厌写文档资料的缘故。于是乎，当仅仅由程序员组成的团队在攻克一个问题时，他们偏向于用代码而不是以文档资料来表示其解法。他们更宁愿匆匆忙忙地一开始就大写特写代码，而不是首先制订一个规格说明书。 在设计阶段发现问题时，可以通过编辑几个文本行而容易地修复它们。一旦代码写完，纠正 错误所付出的代价会高得出奇，这包括情感（人们通常都厌恨丢弃代码的行为）与时间两方 面的原因。因此，实际消除问题是有阻力的。不是根据规格说明书开发出来的软件通常会因 设计欠佳而停滞不前，从而使进度失去控制。这个问题好像己经在Netscape浏览器那里出 现过，它开头的四个版本逐渐变得混乱不堪，致使管理层愚蠢地决定丢弃己写成的代码而从 头开始[6]。而后，他们再次在Mozilla上完全重复了这一错误而创建出一个让公司运作逐 渐失去控制的怪物，整整花了几年的时间才推进到a阶段。

按照我提出的娇惯理论，这个问题的一种解决之道是通过将程序员送去参加一个写作方面的强化班[7]而教他们成为不那么凑合的作者。另外一种解决办法是聘用聪明能干的程序管理 人才，由他们负责编写规格说明书。使用这里给出的任何一种方式，都应该强调那条简单的 规则一一 ”没有规格说明书就没有代码！〃 读者可以阅读本书第5~8章，学习关于写规格说明书的所有内容。

8.程序员拥有安静的工作环境吗

大量的文献表明，通过给能干的工作人员提供宽敞、宁静和独立的工作环境，可以获得很高的生产效率。软件管理方面的经典书籍《人件》（Peopleware)全面论述了这类生产效益的 有关情况[1]。

现在来说明它是如何发挥作用的。大家知道，熟练员工一旦做到“顺手”（也称之为进入状态）就工作得最好。这个时候，他们将全部精力集中在自己的工作上而置身于环境之外。他们忘记了时间，因为精力特别集中而焕发出很高的生产率。这正是他们完成所有可做之事的时候。作家、程序员、科学家乃至篮球队员等人士都可以说出什么叫进入状态。

问题在于，进入状态并不是件容易做到的事情。测量结果表明，人们平均起来似乎要经过 15分钟才开始进入效率最高的状态。有时候，如果你比较劳累或者己经在当天做了很多创 造性的工作，那么你就不能进入最佳状态；在剩下的工作时间里，所要做的事情就是把周围 整理

一下、上一上网或者玩玩跳跳棋。

另外一件麻烦事情是被干扰而退出兴奋点却非常容易。噪音、电话、出去吃午饭、不得不驱车5分钟去喝咖啡，以及被同伴打断——被同伴打断尤为突出，这些事情都可以使人退出兴奋区。如果虽然同伴问你一个问题只将工作中断了一分钟，但是这足以让你花半小时的时间 再重新进入状态，那么你的总体工作效率就很严重了。如果置身如咖啡网吧倾向于营造的那 种人多嘈杂的大房间环境之中，外加几个市场推销员在程序员旁边喋喋不休地聒噪，那么生 产效率将随着熟练员工一次又一次地被打断而打了水漂，你永远也进入不了状态。

对于程序员来说，这尤其很难。生产效率与能够立即改动脑子里短期记忆的许多细节有关。任何类型的中断都能够使这些细节尽皆消失。在重新恢复工作时，任何细节（像正在使用的局部变量名或者着手实现搜索算法的地方）都想不起来了，从而只得再次把它们找出来，这会极大地降低工作速度，直到工作重新变得顺手为止。

它涉及的代数知识并不复杂。毫不客气地说（正如有证据似乎表明的那样），即便是打断程序员一分钟，我们其实打掉了他15分钟所创造的效益。作为这方面的一个例子，让我们将 程序员Jeff与Mutt彼此紧挨着安排在某标准Dilbert肥牛场中一个开放的小卧室里。Mutt 记不住Unicode版本的strcpy函数的名字。他可以查询该函数，这要花费30秒钟的时间；他也可以询问Jeff，这得用去15秒钟的时间。既然紧*Jeff坐着，他选择询问Jeff。不 过，Jeff因此显得心烦意乱而丢掉了 15分钟的产出（仅仅为了节省Mutt 15秒钟）。

现在让我们将他二两人移到有墙有门的独立办公室中去。这个时候，Mutt记不住那个函数 的名字，他可以查询该函数，仍然要花费30秒钟的时间；他也可以询问Jeff，现在得用去 45秒钟的时间并且意味着站起来走一趟(在现有程序员平均身体状况下，很少会那么做！）。 于是，Mutt决定自己去查询该函数的名字。如此一来，虽然现在Mutt损失30秒钟的产出， 但我们为Jeff节省了 15分钟。啊哈！

9.你用到了你资金能力内可买到的最好工具吗

用编译语言编写代码，仍然是在像花园一样多姿多彩的家用电脑上不能立即做成的最后几件事情之一。如果编译过程要用几秒钟以上的时间才能完成，那么使用最新最好的计算机将可以为你节省大量时间。即使编译过程只需要15秒钟，在编译器运行时，程序员也会感到无聊，从而可能转过去阅读《洋葱》[2]，这会使他着迷，于是几个小时的开发效益就没有了。

用单监视器系统调试GUI代码是很头疼的，尽管不是不可能。写GUI代码时，使用两个监视 器会使事情容易得多。

大多数程序员最终都会因为图标或者工具栏而*作位图，并且他们大多没有很好的位图编 辑器可用。虽然试图使用Microsoft Paint来处理位图不啻是个笑话，但这是大多数程序员 不得不去做的事情。

在我最新从事的一项工作中[3]，系统管理员一直不断地给我发送自助午餐肉（信息）。他 抱怨说，我为此占用了服务器上多达220MB的硬盘空间。我指出，以当前的硬盘驱动器价格论，这个空间的花销远远低于我使用的手纸的费用。我哪怕花10分钟来清除我的目录，也 将使生产效率造成惊人的浪费。

拔尖的开发团队是不让程序员遭受磨难的。它会尽可能不让程序员因为使用功能不强的工具而遇到挫折，以免程序员心情烦躁而感到不愉快。烦躁不安的程序员不属于高产的程序员之列。 说到底，程序员是容易通过给他们提供最棒最新的东西而得到满足的。这是一种远比通过支 付极富竞争力的薪水来促使他们为你工作，要廉价得多的途径！

10.有测试人员吗

如果团队没有专门的测试人员（至少为每两个或者三个程序员配备一个测试人员），那么意味着要么分发充满故障的产品，要么通过让价值 30/小时的测试人员完成的工作而浪费钱财。在测试人员方面吝惜钱财谈不上节俭，这种做 法显得不可理喻，我只能是嗤之以鼻地说，较多的人不会认可它。第22章对该内容会有更 多的介绍。

11.新聘人员在试用期间编写代码吗

你会聘用一个魔术师而不叫他露几手戏法吗？当然不会。你会让酒店老板承办婚礼宴席而不品尝他们的菜肴吗？这很难说。（Marge阿姨除外，你要是不让她做她“著名的”碎肝饼， 她会嫉恨你一辈子。）

尽管如此，每天仍然有一些程序员是因为持有令人印象深刻的履历或者面试人员喜欢跟他们聊天而被聘用的。或者他们只是被问了一些非常琐碎的问题（比如说“C「eateDialog〇与DialogBox()之间有什么不同？”），这些问题是可以通过查看文档资料得以作答的。用不 着在意他们是否记住了编程方面成千上万的细枝末节，应该在乎的是他们能否构造代码。甚 或还有更为糟糕的情况，那就是程序员被问了一些“打哈哈”的问题一一这些问题如果你知 道答案似乎显得很容易，可要是你不知道答案，要回答出来就显得极为不可能。

务请停止这样的做法。诚然，你可以在面试期间做想做的任何事情，但是一定要让应聘者编写一些代码（关于这方面的更多建议，可阅读我的《面试游击战指南》一文，本书的第20 章收录了该文。）

12.进行走廊可用性测试吗

走廊可用性测试指的是，在走廊里随便抓一个从身边走过的人，要他试着使用你所编写的代码。如果对5个人进行了这类测试，那么就可以了解隐藏在代码中的95%的可用性问题。

好的用户界面设计并不像你想像的那样难，并且，它对于顾客喜爱与购买你的产品至关重要。 我写的关于用户界面设计的书[4]集中叙述了这方面的内容，它是为程序员所写的一个初级 读本。

不过对于用户界面最为重要的事情是，如果将程序交给一部分人（事实上，5~6个人就够了）看，那么你会迅速发现人们掌握着最大的问题。关于这方面的原因，可阅读Jakob Nielsen 的有关文章[5]。即使你的U I设计技能很缺乏，但是只要强迫自己进行走廊可用性测试（这 用不着付出什么代价），那么设计出来的UI将是非常非常好的。

使用Joel测试的四种途径

1. 评价自己的软件机构，并说出是如何评价的，这样我可以与你交流一下了。
2. 如果你是程序设计团队的经理，那么使用该测试作为校验表，以保证团队确实在尽可能地发挥作用。如果一开始就评定一个团队为12分，那么可以集中精力不使业务人员去打扰 程序员而使他们置身其外[6]。
3. 如果你在决定是否接受一项编程活件的话，那么询问你未来的雇主在该测试上的得分情况如何。如果测试的分值太低，那么就要确保自己有权威去修正这种局面。否则，你会面临 失败而徒劳无益。
4. 如果你是一位正在努力评判某个程序设计团队的价值的投资人，或者你的软件公司正在考虑与另外一家公司合并，那么这个测试可以提供一个快速而简易的尺度。

//后记：2004年6月14日 //自从Joel测试在2000年8月推出以来，我从世界各地的开发人员那里收到了成堆的汇报 各自团队如何评分事宜的 //电子邮件。尽管结果的分布显得相当均匀，我还是要说，绝大多数分值处在2与3之间的 一个位置。嗯！我也收到 //一些在无组织的“枪手”软件开发实践当中成长起来的开发人员的来信，他们拒绝从那些 看起来在Joel测试方面 //天生只能得低分的公司招揽新活。并且，我还收到一些团队经理给出的杰作，他们以Joel测试作为提高团队素质 //的进阶过程而推进自己的管理工作。 //与此同时，似乎许多软件开发组织已经远远地走出了 Joel测试的范畴，而进入复杂的官 僚政治动脉硬化苦斗之中 //。什么时候会出现这种情况是很容易说出的，因为人们在准备开会方面所花的时间比开发 软件更多。在Joel测试 //上得到12分的不错成绩，然后就被多如牛毛的政治活动纠缠得脱不开身而做不了什么事 情，这是很有可能发生的 //一种情况。 //没有必要告诉别人是我说了些什么，不过，自上世纪90年代初我在微软公司工作的那个 时期以来，所有证据都表 //明，该公司或多或少地搁浅乃至停滞均是因其过于庞大的规模、内部政治纷争与官僚系统 的不检点所致。还需要 //证据吗？ //Tablet PC 一一这是一种只有微软公司的中层经理们才可能喜爱的东西一一似乎设计成了 让华盛顿Redmond的程 //序经理们可以开一整天的会，却不至于使其日常电子邮件处理事宜显得过于滞后。这不单 纯只是微软才存在的现象。 //如果突然间地发现自己在安装与配置巨无霸软件开发系统或“什么什么可视化企业体系设 计大师”之类的软件方 //面，花费的时间太多，或者说要开始开发产品时还在团队重新学习“极限编程（Extreme Programming)”与 //UML方面反反复复而让他们脑袋变大，那么即使在Joel测试方面真的做得不错，你还是 可能陷入困境。

[1]见Tom DeMarco与Timothy Lister合著的《人件:高产项目与团队》第2版（Peopleware: Productive Projects and Teams, Second)，该书由 Dorset 出版社于 1999 年出版发行。

[2]见 http://www.theonion.com 。

[3]见第32章。

[4]Joel Spolsky 著的《程序员用户界面设计KUser Interface Design for Programmers)(Apress于2001年出版）。该书的一部分可以从 http://www.joelonsoftware.com/uibook/chapters/fog0000000057.html 上免费使用。

[5]Jakob Nielsen写的“为什么只需测试5个用户”一文于2000年3月19日在.useit.com 上发表。详细内容见 www.useit.com/alertbox/20000319.html 。

[6] www.joelonsoftware.com/articles/fog0000000072.html 。

四、每一位软件开发人员必须、绝对要至少具备 UNICODE 与字符集知识（没有任何例外！）

有过这样的经历吗？从保加利亚的朋友那里收到电子邮件，主题却为“？？？？？？？？？？？？？？？？？？？”？

我一直十分沮丧地试图去揭示到底有多少软件开发人员没有真正跟上那个充斥着字符

集、编码技术、Unicode及其素材的神秘世界的变化速度。数年以前，FogBUGZ[1]的P测试人员一直想知道自己是否能够处理日语电子邮件。日语？他们收到过日语电子邮件吗？我不知道。

当近距离地审视我们一直用来解析MI ME电子邮件的商业ActiveX控件时，我发现它确实 在字符集方面做得很差。实际上，我们只得用自己写的代码取消它完成的错误转换，然后重 新进行转换工作。在打量另外一个商业软件库时，我们发现它同样有一个毛病很多的字符代 码实现形式。我与那个软件包的开发人员进行了联系，他好像认为“不能去为它做任何事情”。 与许多程序员一样，他仅仅希望它会以某种方式滑过去就行了。

不过，它是不会过去的。我发现流行的Web开发工具PHP，几乎完全忽视了字符编码方面 的问题[2]而单纯地使用8位字符，它不大可能开发出好的国际Web应用程序。我真正感觉到，够用是怎么一回事情了。

因此，我要做出一项宣示：如果你是一位21世纪的程序员，却不知道字符、字符集、编 码技术与Unicode方面的基本知识，那么要是让我逮着，我打算通过让你在潜艇上剥6个月的 洋葱来惩罚你。我发誓我会这么做。

还有一件事情：

它并不那么难。

在本章中，我将告诉你到底每个真正动手编程的程序员应该知道什么。所有关于“纯文 本=ASC 11=字符是8位”之类的看法不仅是错误的，而且错误得令人绝望。要是你仍然按这样 的方式去编程，那么你比一位不相信细菌的医学博士好不了多少。务请在读完本章之前不要 再写另外一行代码。

在实际开始之前，我得提请你注意，如果你不是罕见的几位知道一些国际化知识的人员 之一，那么你会觉得我整个讨论有点儿过于简单。其实，我不过是在这里试图做最少的一点 抛砖引玉的工作，以便让大家都能够理解后续的知识，并且能够使写出的代码有望在任何语种的文本环境，而不仅仅是不包括重音的英文子集的环境下使用。

我还要提醒大家注意，尽管字符处理仅仅是创建能够在国际语言环境下使用的软件所要 完成的很微小的部分，不过我一次只能写一件事情，因此，今天在这里谈到的就是字符集。

历史情景

理解这个素材最为容易的方式是按年代顺序依次推进。你可能觉得我打算在这里谈论像EBCDIC之类的古老的字符集。唔，不会的。EBCDIC与你的生活毫不相干。我们也没有必要追 溯那么久远的年代。

在计算机出现后的中期时代，人们开发了UNIX，K&R写出了《C程序设计语言》，一切都 显得非常简单。EBCDIC才刚刚浮出水面。惟一很要紧的字符集就是有效而古老的无重音英语字母，而人们为这些字符开发了一套称为ASCII的编码，它能够使用32到127之间的数字表示各个字符[3]。空格的编码是32,字母A的编码是65,等等。这种编码能够很方便地用七个二 进制位来表示。

由于那个时期生产的大多数计算机使用8位大小的字节，因此用户不仅可以存放所有可 能的ASCII字符，而且有整整一位空余下来。如果你技艺高超，可以将该位用做自己离奇的 目的:WordSta「中那个发暗的灯泡实际上设置这个高位，以指示一个单词中的最后一个字母，同时这也宣示了 WordSta「只能用于英语文本。码值小于32的代码称为非打印字符而作为杂项 字符使用，也就是当做小不点儿来用。这些字符用做控制字符，比如，码值为7的字符使计算机发出“嘟嘟”声，而12对应的字符使当前走一页打印纸并换入新的一页。

对于一个说英语的人而言，这一切都是不错的。

由于字节有多达8位的空间，因此许多人在想：“呀！我们可以把128~255之间的编码用 做个人的应用目的。”问题在于，同时产生这种想法的人相当多，而且在128~255之间的各 个位置上应该存放什么这一问题上，真是仁者见仁智者见智。IBM-PC有一种OEM字符集，它 提供了一些欧洲语系的重音字符，以及一组画线字符一一水平线条、竖直线条与右边有弯折

的水平线条等。你可以使用这些画线字符在屏幕上画出好看的方框与线条，只要环境清洁干燥，绘制的图形在8088计算机上仍然看得见。

事实上，只要人们开始在美国以外的地方购买计算机，那么各种各样的不同OEM字符集 都会进入规划设计行列，并且各人都会根据自己的需要使用高位的128个字符。比如，在一 些PC机器上编码为130的字符显示为而在以色列销售的计算机上它显示为希伯来语的第三 个字母A。因此，当美国人想把自己的履历（「gsum6s)发给以色列时，显示出来的会是r 入sum AS。在使用诸如俄语之类的语种的多种情况下，对高端128个字符可能存在很多不同 的处理思路。如此一来，甚至在同语种的文档之间就不容易实现互换。

最后，这个人人参与的OEM终于以ANSI标准的形式形成文件。在ANSI标准中，每个人都 认同如何使用低端的128个编码，这与ASCII相当一致。不过，根据所在国籍的不同，处理编 码128以上的字符有许多不同的方式。这些不同的系统称为代码页[4]。这样一来，比如说在 以色列DOS使用称为862的代码页，而希腊用户使用的代码页是737。这两个代码页在128以下是一样的，但在128以上则不相同，该代码段存放的是一些古怪的字母。MS-DOS的国家版本有几十个这样的代码页，负责将所有的英语信息转换成冰岛语。它们甚至还拥有几个“使用 多语种”的代码页，从而可以在同一台计算机上处理世界语与加利西亚语！呜哇！不过，要 说一

下，在同一台计算机上处理希伯来语与希腊语是完全不可能的事情，除非自己编写一个 定制程序以显示使用位映象图形的任何内容，因为希伯来语与希腊语需要能够对高端编码做 出不同解释的不同代码页。

同时，甚至更为令人头疼的事情正在逐步上演，亚洲国家的字符表有成千上万个字符， 这样的字符表是用8位二进制无法表示的。该问题的解决通常有赖于称为DBCS(double byte character set，双字节字符集）的繁杂字符系统。这个字符系统将一些字符存储为一个字节，而让另外一些字符占据两个字节。虽然在字符串中向前移动很容易，但向后移动几乎不可能做得很棒。我们不鼓励程序员使用s++与s–的代码形式在字符串中前后移动，而是提 倡调用Windows的AnsiNext与AnsiPrev函数，这两个函数知道如何去应付所有的繁杂局面。

不过，仍然需要指出一点，多数人还是姑且认为一个字节就是一个字符，以及一个字符 就是8个二进制位，并且只要确保不将字符串从一台计算机移植到另一台计算机，或者说一 种以上的语言，那么这几乎总是可以凑合。当然，只要一进入Internet从一台计算机向另一台计算机移植字符串就成为家常便饭了，而各种复杂状况也随之呈现出来。令人欣慰的是，Unicode随即问世了。

Unicode

Unicode勇往直前地创建一种单一字符集，试图囊括地球上所有合理文字体系，以及诸 如一些Klingon之类的人为书写体制。一些人错误地认为，Unicode就是一种每个字符占用16 个二进制位，从而总共可以表示65 536个可能的字符的16位字符编码方案。这并不十分正确。 它是关于Unicode的惟一最为流行的神话，因此，要是你也这样想，大可不必感到难过。

事实上，Unicode在考虑字符方面给出了一种不同的思路。你需要去理解它的这种思考 方式，否则一切都显得毫无意义。

直到现在，我们一直认定字母映射为磁盘或者内存的某些位：

```
A -> 0100 0001
```

在Unicode中，一个字母映射为一种称做代码点（code point)的对象，它仍然只是一 个理论上的概念。该代码点如何在磁盘或者内存中进行表示完全是一种微妙的个体行为。

在Unicode中，字母A是一个柏拉图式的理想，它只能飘荡在天国之中：

```
A
```

这个柏拉图式的A不同于B，也不同于a，但与A或者或者A却是一样的。认为Times New Roman字体的A与Helvetica字体中的是相同的字符，但不同于小写字母a的想法，看起来并 不会引起很大的争议，但在某些语言中仅仅指出字母是什么完全可能引起争议。德语的字母 б是真正的字母，抑或仅仅是ss的一种花样书写形式？如果一个字母的形状在单词的末尾变 化

了，那么它是一个不同的字母吗？希伯来语系认为是，而阿拉伯语系则说不。不管怎样， 聪明的Unicode人在最近大约十年里一直在设法处理这个方面的问题，与此相伴的是大量高 度政治化的争论。不过，你用不着担心，他们已经把它全部处理妥当了。

Unicode组织为各种字母表中每个理想化的字母分配一个神奇的编号，其书写形式为 U+0645。这个神奇的编号称为一个代码点。U+的含义是“Unicode”，这些数字是十六进制的。U+0639表疋阿拉伯字母Ain。英语字母A是U+0041。使用Windows 2000/XP的charmap工具 程序或者访问Un i code网站，可以找到所有的字母及其编码[5]。

Unicode能够定义的字母个数其实没有限制，实际用到的总字符个数已经远远超出了65 536的范围。可见，并不是每个Uni code字母能够真正挤压成两个字节，不过，这终归还是一 个神话。

好了，我们来看一个字符串：

```
Hello
```

该字符串在Uni code中对应如下形式的5个代码点：

```
U+0048 U+0065 U+006C U+006C U+006F
```

这不过是一簇代码点。其实，就是一些编号。到此为止，我们还没有谈到如何在内存中 存放这类字符串或者怎样在电子邮件中表示它们。

[1]这是我们的故障跟踪产品，见www.fogcreek.com/FogBUGZ。

[2]见ca3.php.net/manual/en/language.types.string.php。

[3]关于ASCII字符的更多信息，参见 www.robelle.com/library/smugbook/ascii.htmlo

[4]关于代码页的更多信息，参见 www.i18nguy.com/unicode/codepages.html#msftdos〇

[5]见www.unicode.org。

编码

是给出编码知识的时候了。

关于Unicode编码，最为容易同时也因此导致2字节神话出现的想法是：“嗨，让我们将 这些编号各自用两个字节存放吧！”于是，”Hello”变成了 `00 48 00 65 00 6C 00 6C 00 6F` 对吧？不要这么性急！可以存为 `48 00 65 00 6C 00 6C 00 6F 00` 吗？

喏，从技术上讲是可以的，我就相信这一点。事实上，早期的实施人员就想以高端或者 低端两种存放模式在内存中表示Unicode码。不管特定的计算机在何种模式下运行最快，喔， 也不管是傍晚还是清晨，现在已经有了两种存放Unicode码的方式。于是，人们被迫提出在 每

个Unicode字符串的开头存放一个FEFF标记的怪异协定。这个标记称为Unicode字节顺序 标记（Unicode Byte Order Mark)。要是交换了它的高低位字节，该标记就成为FF FE。这 样一来，阅读字符串的人就知道必须将字符串的每两个字节进行一次交换[1]。吁！苍茫世 界当中不是每个Unicode字符串都会在开头放一个字节标记。

曾几何时，这样的做法好像是足够好了。但是，程序员却一直没有停止抱怨："瞧瞧所 有那些零！"要知道，作为美国人，他们阅读的英语文本很少用到U+00FF以上的代码点。 还有，他们是来自加利福尼亚州的无拘无束的嬉皮士，保守与冷嘲热讽是他们的天性。如果 他们是得克萨斯人，就不会在乎要狂吃双份的字节。不过，那些咕咕叽叽的加利福尼亚人不 会忍受将字符串占据的存储空间加倍的想法。而且，不管怎么说，已经存在的一些该死的文 档都赫然使用着各种各样的ANSI与DBCS字符集，谁会去将它们全部转换过来呢？仅仅是这个 原因，大多数人决心在几年之内对Unicode视而不见。不过，与此同时，情况却变得尤为糟 糕。

于是乎，光彩照人的UTF-8[2]概念就横空出世了。UTF-8是另外一种存放字符串的 Unicode代码点的体系，它在内存中使用8位字节存放那些神奇的U+编号[3]。在UTF-8中，从 0~127的每个代码点存放在单个字节里面，只是128以上的代码点才使用2个字节、3个字节， 乃至多达6个字节的空间来进行存储。

最小十六进制	最大十六进制	二进制字节序列
00000000	0000007F	0vvvvvvv
00000080	000007FF	110vvvvv 10vvvvvv
00000800	0000FFFF	1110vvvv 10vvvvvv 10vvvvvv
00010000	001FFFFF	11110vvv 10vvvvvv 10vvvvvv 10vvvvvv
00200000	03FFFFFF	111110vv 10vvvvvv 10vvvvvv 10vvvvvv 10vvvvvv

这具有使英语文本看起来在UTF-8与ASCII中显得完全一样的边界净化效应。如此一来， 美国人用不着在意有什么地方不对了。只是世界上其他地方的人得从铁箍中跳过去。具体地 说，编码为 U+0048 U+0065 U+006C U+006C U+006F 的 "Hello〃字符串将存储为48 65 6C 6C 6F。瞧！这与地球上存储为ASCII、ANSI与各种OEM字符集的形式完全相同。

现在，如果你非得固执地使用重音字母或者希腊字母或者Klingon字母，那么就必须使 用几个字节来存放单个代码点，但是美国人永远不会意识到有什么异样。（UTF-8同样具有 后面这种不错的属性：现有的那种想用单个0字节作为空终止字符的陈旧而愚昧的字符串处 理程序，不会截掉字符串。）

到此为止，我已经说出了Unicode的三种编码方式。将编码存储为2个字节的传统方法称 为UCS-2 (因为它有2个字节）或者UTF-16 (因为它有16位），不过仍然需要弄清它是高端 存放模式的UCS-2,还是低端存放模式的UCS-2。还有一种流行的新UTF-8标准，它同样可以 很自然地使用，要是你碰巧处理令人满意的英语文本，并拥有完全不知道世间除了 ASCII之 外还有其他何物的死脑筋程序的话[4]。

其实，还有一些其他的Unicode编码方式。一种称之为UTF-7的编码体系与UTF-8有许多 相似之处，只不过它保证高位总是为0。这样一来，要是你必须通过某类极权国家的严密电 子邮件系统传递Unicode，它可以在高压之下仍然毫发无损。这种电子邮件系统认为用7位二 进制表示一个字符完全够用了，谢谢。另外一种称为UCS-4的编码方式用4个字节存放一个代 码点，它虽然具有这样一种良好性能：将各个代码点都存储为个数相同的字节，不过，天哪！ 即使得克萨斯人也不会鲁莽到浪费那么多的内存。

事实上，既然以Unicode代码点形式的柏拉图理想字母来思考问题，Unicode代码点也就 能够按任何守旧的编码方案进行编码了！比如，可以用ASCII编码方案、老式OEM希腊编码 方案、希伯来ANSI编码方案或者几百种现有编码方案之一来编码Unicode字符串“Hello”(U+0048 U+0065 U+006C U+006C U+006F)，只是要抓住一点：一些字母可能是不出现的！如果在试图使用的编码方案中没有相应Unicode代码点的等价内容，那么通常会显示一个小问号“？〃，或者更为先进一些的话，就显示一个方框。

传统编码方式不下几百种，不过它们仅仅能够正确地存放一些代码点，将所有其他的代 码点转变为问号。流行的英语文本编码方案有Windows-1252 (西欧语种的Windows 9x标准） 与ISO-8859-1,即Latin-1 (同样对任何西欧语种都有用）[5]。不过，如果试图用这类编码 方式存放俄语或者希伯来语字母，那么得到的结果就会是一簇问号。UTF 7, 8, 16与32都具 有能够正确存放任何代码点的优良特性。

[1]关于字节顺序标记的更多信息，参见 msdn.microsoft.com/library/default.asp?url=/library/en-us/intl/unicode_42jv.asp。

[2]见 www.cl.cam.ac.uk/~mgk25/ucs/utf-8-history.txt 。

[3]关于UTF-8的更多信息，参见 www.utf-8.com/ 。

[4]见 www.zvon.org/tmRFC/RFC2279/Output/chapter2.html 。

[5]关于 ISO 8859-1 字符集的概况，参见 www.htmlhelp.com/reference/charset 。

编码中惟一最重要的事实

即使你将我刚才所说的一切忘得一干二净，我也要请你记住一个极其重要的事实。有一 个字符串，而不知道它所使用的编码方案是毫无意义的。你再也不用将脑袋紧贴在沙地上而 假想“纯”文本就是ASCII。

纯文本不是这么一回事

如果在内存、文件或者电子邮件中有一个字符串，那么应该知道它使用的是什么编码方 案，否则就不能将它正确地解释或者显示给用户。

其实，几乎所有诸如“Web网页好像是乱码”或者“在使用重音字母时无法阅读电子邮 件”之类的愚蠢问题，都可以归结到天真的程序员身上，他不了解这样一个简单的事实，即 如果不告诉我某个特定字符串是用UTF-8，或者ASCII，或者ISO 8859-1 (Latin 1)，或者 Windows 1252 (西欧）编码的，那么我就不能正确地显示，乃至弄清楚它在什么地方结束。 编码方案数以百计，而在代码点127以上，拍脑袋已经没有效果了。

如何保存字符串用到的编码方案信息？喏，做这件事情有许多标准的方式。对于电子邮 件信息，需要在表窗体标题放一个字符串

```
Content-Type: text/plain; charset="UTF-8"
```

对于Web页，最初的想法是Web服务器通过网页本身返回一个类似于Content-Type的http ——不是在HTML当中而是作为一个响应标题在HTML页面之前返回。

这会引起一些问题。假设一个大型Web服务器拥有许多站点，成千上万的网页是由许多 人使用多种不同语言发布的，网页使用任何在Microsoft FrontPage看来很适合产生的编码 方案。由于Web服务器实际上不知道每个文件究竟是用什么编码方案写成的，因此它不能发 送Content- Type 标题。

如果能够使用某种特殊标记将HTML文件的Content-Type内容自然地放在HTML文件当中， 会是很方便的。当然，这会让纯化论者发疯。不过，在知道HTML文件使用了什么编码方案之前，如何读取HTML文件呢？！幸运的是，几乎每个用得很普遍的编码方案，对码值32与127之间的字符都以相同方式进行处理。因此，人们总是可以充分展示HTML页而不必使用一些很稀奇的字母：

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
```

不过，这里的meta标记确实位于段中很*前的位置。因为只要Web浏览器看到该标记，它就会停止解析页面，并在使用指定的编码方案重新解释整个页面之后进入下一轮。

如果Web浏览器在http报头与meta标记中都找不到任何Content-Type,该怎么办呢？ Internet Explore「确实做了件非常有趣的事情：基于不同语言以典型编码方案得到的典型 文本中各种字节出现的频率，试图猜出使用了什么语言与编码方案。由于各类不同的旧8字 节代码页倾向于将其母语字母放在128与255之间的不同范围中，并且人类使用的每个语种具 有不同特征的字母使用概率分布，因此使这种做法确实有了发挥作用的机会。是的，它确实 显得有点怪异。不过，这种做法通常能够满足那些从不知道要使用Content-Type报头的幼稚 网页编写人员，在Web浏览器中阅读网页的需要，并且看起来还很不错。

直到有一天，编写的网页确实与母语的字母使用概率分布情形不一致，Internet Explore「则认定它是朝鲜语而加以显示。在我看来，这表明了一点，Jon Postel的格言“宽 进窄出”实在不是一条好的工程原则[1]。不管怎么说，一旦用保加利亚语写的网站却以朝 鲜语的形式出现

（甚至是毫不相干的朝鲜文），可怜的读者该怎么办呢？使用菜单“View | Encoding (视图|编码）”试图应用一组不同的编码方案（东欧语种可用的编码方案至少有 一打），直到图片能够较为清楚地显示出来为止。即使这位仁兄知道如此行事，可大多数人 却不知道这一招。

对于由本人的公司Fog Creek Software发布的Web站点管理软件最新版本CityDesk[2] 而言，我们决定在内部用UCS-2 (2字节）Unicode处理一切。这种编码形式也是VisualBasic、 COM与Windows NT/2000/XP作为其固有字符串类型使用的。在C++代码中，字符串就定义为wchar_t而不是char，并且使用以wcs打头的函数而不是以st「打头的函数（比如说，使用wcscat与wcslen，而不是strcat与strlen)。要用C代码生成UCS-2文本字符串，就得像L”Hello”—样在字符串前面放一个L。

CityDesk发布Web页时，将网页转换为UTF-8编码形式，该编码方案多年来一直得到Web 浏览器的良好支持。“Joel on Software ”（《Joel说软件》）的29种语言版本全部使用这 种方式编码，我迄今为止还没有收到一个人的来信说他在浏览网页内容时遇到任何麻烦[3]。

本章写得有点长，并且未能覆盖需要了解的一切字符编码与Unicode内容。但我还是希望你好好地读一读，它对于你回过头来看编程有一定好处，正如用抗生素，而不是猛药与符 咒。这就是我现在留给你去做的一项任务。

[1]Jon Postel的话，引自信息科学学院的1981年9月发布的文档“RFC 791 - Internet Protocol”。

[2]见 www.fogcreek.com/CityDesk 。

[3] www.Joelonsoftware.com/navLinks/OtherLanguages.html 。

五、轻松写就功能规格说明书 - 第1节：为什么烦心？

当约耳试验刚发表时，读者反映说执行上最困难的一点就是写规格.看起来规格 就像用牙线：大家都知道应该要，却没有人真的在做.

为什么大家不写规格呢?有人声称跳过写规格这个步骤可以节省时间.讲得好像 写规格像是什么奢侈品，只有NASA航天飞机工程师或为大保险公司工作的人才 能做.胡说八道.不写规格是软件项目中一个最大且不必要的风险.这就像只在背 上挂件衣服就妄想飞越摩哈维沙漠一样的愚意.那些不订规格直接写程序的程序 员和软件工程师都会自认是神枪手，可以朝背后开枪.才怪.他们根本没有生产力. 他们写出烂程序并做出低劣的产品，而且还让项目承受完全可以避免的极大风 险.

我认为所有重要的项目（写程序时间超过一周或一个以上的程序员）都一样，如 果没有规格，绝对会耗时更久并且写出质量低劣的程序.理由如下.

规格最重要的功能是要设计程序.即使你是自己一个人写好所有的程序，而且 写了规格也只是自己在看，写规格这个动作(详细描述程序运作)还是能迫使你 实际设计程序.

让我们看看分属两家公司的两名虚构程序员.快餐香蕉软件的史快手从来不写 规格规格？我们才不要讨厌的规格！”而好脾气软件公司的罗杰先生则在规格 确定前绝对不写程序.他们只是我众多虚构朋友中的两位.

史快手和罗杰先生有个相同点：他们都负责自家产品2.0版的向后兼容事宜.

史快手决定，提供向后兼容的最佳方法就是写一个转换器，把1.0版的档案转成2.0 版档案.她开始打程序，喀嗒喀嗒，硬盘转呀转，鸡飞狗跳.大约两周后弄出一个还 不错的转换器.不过史快手的客户不太高兴.史快手的程序等于强迫全公司一次 升级成新版本.史快手最大的客户Nanner Splits无限公司拒绝购买新软件.他 们要确定2.0版软件能不经转换使用1.0版档案.史快手决定写一个回溯转换器 并挂在〃存盘〃功能内.这就有点复杂了，在使用2.0版的功能时似乎一切正常， 不过等要把档案存成1.0格式时就有问题了.你会发现前半小时用的新功能都无法以旧的文件格式储存.所以回溯转换器又写了两周，而且也不是很好用.这样 子4周就过去了

而好脾气软件公司的罗杰先生是那种一切照规矩的人，没拿到规格前是绝对不 写程序的.他花了20分钟设计出和史快手一样的向后兼容功能，并且写出大意 如下的规格：

如果开启的档案是由旧版产品所建立的，就把档案转成新格式.

把规格先拿给客户看看，客户说话了〃等一下！我们不想让全公司都升级！”所以 罗杰先生再想想后把规格改成：

如果开启的档案是由旧版产品所建立的，就把档案在内存内转成新格式.等要 储存这个档案时，使用者可以选择存回旧格式.

这样又用了 20分钟.

罗杰先生的老板是个面向对象狂.他看看这份规格,觉得有点问题并且建议用另一种架构。

把程序重组成使用两组接口 ： V1和V2.V1包含所有版本一的功能，而V2则是继承V1 并加上所有新功能.这样V1::Save可以处理向后兼容事宜而V2::Save可以储存 所有新东西.如果你开启V1档案后使用到V2的功能，程序可以马上警告你，你必 须转换成新文件格式或放弃使用新功能.

又20分钟去了。

罗杰先生不太高兴.因为重组程序要花3周而不是他原先估的2周！不过的确优雅 地解决了客户所有的问题，所以他还是回去照做了.

罗杰先生所花的时间总共是3周加1小时.史快手则花了4周，不过史快手的程序 没那么好.

这个故事告诉我们，只要举的例子够怪，什么事都可以证明.对不起,刚才是开玩笑 的，我不是这个意思.这个故事真正要说的是，当你以人用的语言设计产品时，只 需要花几分钟就能考虑多种可能并且修订及改进自己的设计.没有人会觉得在 字处理器里删掉一段有啥大不了的.不过当你用程序语言设计产品时，反复设计 就得花上好几周了.更糟糕的是，某段程序可能只花程序员2周就写出来,可是他 会一直死抱着那段程序，不管程序错得多么离谱.虽然史快手的程序并非最佳的 架构，可是不管她老板或客户说什么，都无法说服她把那段漂亮的转换程序丢 掉.结果最终的产品很容易变成最初设计和错误设计以及理想设计间的妥协.它 是"我们所能做到的最佳设计，因为这些程序都写好了，而我们又舍不得把写好 的程序丢掉重写".如果光是〃我们所能做到的最佳设计"，没有后面那些废话会 更好.

所以这就是写规格的天字第一号理由.天字第二号是节省沟通时间.当你有写规 格时，对于程序应有的作用只需要讲解一次.团队里其他人只要读规格就好了. 品保人员读了规格就知道程序的动作同时也知道如何测试.营销人员读了规格

就能写出暧昧的资料，放在网站上宣传尚未出现的产品.事业开发人员没好好 读规格却在幻想，以为产品能治疗秃头肿瘤还有毒瘾，不过这样能吸引投资者, 所以没啥关系.开发人员读了规格就知道要怎样写程序.客户读规格则是要确定 开发人员正在制作他们想买的产品.技术作者读了规格才能写出好的手册(然后 被遗失或丢掉,不过这是另一个故事）.经理也读规格，才能让自己好像知道经 营会议的内容.如此类推.

这所有的沟通在没有规格时还是会发生，因为这都是必要的，不过这时的沟通 都是特别安排的.品保人员无论如何都得玩这个程序，看到有任何不对劲就会再 度中断程序人员，并且问另一个程序应该怎么动的蠢问题.这不只会毁掉程序 人员的生产力，而且程序人员也习惯照自己程序的写法回答问题，而不会回答" 正确的答案".结果品保人员实际上是用程序来检验程序，而非以设计来检验程 序，而后者应该会好一点吧.

当你没有规格时，可怜的技术作者会遇到很好笑的事（以极悲惨的方式）.技术 作者通常没有中断程序人员的特权.在很多公司中，如果技术作者习惯中断程 序人员问些程序运作上的问题，程序人员会找经理告状，说这些该死的作者让 他们什么事都做不了，请他们闪远一点.经

理为了想提升生产力，就禁止技术作 者不要再浪费程序人员宝贵的时间.这种公司很容易分辨，因为它们的说明档 和手册里讲的没比画面上多多少.当你在画面上看到以下的讯息 你要启动LRF-1914支持吗？

…当你点选”说明”就会看到一段叫人哭笑不得的内容：允许你选择LRF-1914支持(默认值)或无LRF-1914支持。如果你需要LRF-1914支持，点选〃是〃或按”Y”.如果不需要LRF-1914支持，点选〃否〃或按”N”。

嗯，谢了.很显然技术作者试图掩饰他们不知道LRF-1914支持是啥的事实.他们 不能问程序人员，因为(a)他们太害羞了，或是（b)程序人员在海得拉巴（印度） 而他们在伦敦，或者(c)管理阶层禁止他们中断程序人员或是其他各种数不清的企业弊病，不过根本的问题在于没有规格.

写规格的天字第三号理由就是没有详细规格就无法订出时程.有些场合没有时程 是可以的,例如你打算花14年研究的博士论文，或打算写下一版等好了就会推出 的毁灭公爵游戏.不过几乎所有真实的事业都必须知道事情需要多久完成，因 为开发产品是要花钱的.你不可能不知道价钱就买牛仔裤，那么一个负 责的企 业怎么能不知道要多少还有要多少钱，就决定是否制作某个产品呢？想多了解 时程安排，请参考无痛软件时程.

有个很糟糕的常见错误，就是对某个项目的设计争论不休，结果永远无法排除 争议.Windows 2000的首度开发人员Brian Valentine就有句名言”10分钟内给 你决定，否则下一个免费”（译注:美国著名的披萨连锁店达美乐有一句广告标语 “30分钟内送达，否则下一个免费”.同样的,Brian会在10分钟内决定，否则会免费 给你下一个决定,不过要老板下决定本来就不用付钱给你的，不是吗?）

在很多的程序开发组织中，每次出现设计争议总是没有人愿意作出决策(通常都 是因为政治理由）.所以程序人员只去做那些没有争议的项目.等时间消逝而所 有困难的决定都会压到最后才决定.这些项目很可能会失败.如果你针对某项新 技术开了家新公司，却注意到公司有结构性问题无法作出决策，你最好关门大 吉把钱退还给投资者.因为你永远都无法推出产品.

写规格是处理这些烦人设计决策的好方法，这些问题有大有小，不过没写规格就 看不到.即使很小的决策问题也可以写规格搞定.举例来说，如果你正在建一个 会员网站，你们可能都会同意，当用户忘记密码时应该发信告知密码.很好， 不过对写程序来说还不够.要把程序写出来得知道邮件中实际使用的文句.大部 份公司都不认为程序人员能撰写用户实际接触的文句（而且通常都有很好的理 由）.所以会要求营销人员或公关或是其他文学素养好的人决定讯息的文句内容.〃王大明先生，这是阁下忘记的密码.希望阁下往后能小心一点.〃当你强迫自己 写出一份良好而完整的规格时(我很快就会详述这一点），就会注意到这所有的项目并且把它解决，或者至少标示出来以便修正.

好了.我们现在应该都能同意.规格是一切的根本.我怀疑大部份的人是否懂这一 点,另外我讲的虽然有趣，却不知是否真能让你学到新东西.那么为什么人们不写 规格呢？绝不是省时间，因为根本不会省，而且我认为大多数程序员都知道(在 大部份组织中，唯一存在的”规格”都是断简残篇，是程序员写好程序后解说某 该死功能第三百遍后，用记事本打出来的一页文字文件.

我认为真正原因是多数人都不喜欢写东西.瞪着空白的屏幕会让人感觉极度地 沮丧.我个人克服写作恐惧的方法是去学校参加写作课，一星期写3到5页的短 文.写作就像肌肉一样，写得愈多就愈能写.如果你必须写规格又写不出来，开 个期刊定讯,建一个weblog，上个创作写作课程，或是写封信给亲戚或四年未 联络的大学室友.任何把文字填到纸上的活动都能增进你写规格的技巧.如果你 是个软件开发经理而该写规格的人写不出来，把他送到山里面上两周创作写作 课吧.

如果你未曾在一家有写功能规格的公司做过，可能会没看过功能规格.我会在 本系列的下一篇中展示一篇简短的规格范例供你参考，另外我们也会讨论一份 好的规格必须具备的元素.

六、轻松写就功能规格说明书 - 第2节：什么是规格说明书？

(你看过第一篇吗？如果还没有可以到这裡看。)

这一系统的文章是在探讨功能规格而非技术规格(译注：也称为工程规格)。人们常会把这两者搅混。我不知道是否有其他标准术语，不过我个人的用法如下：

功能规格纯粹由使用者的角度来描述产品如何运作。它不管东西是怎麼做出来的。功能规格会述及功能，还会详述画面、功能表、对话框等等。

技术规格则是描述程式内部的实作。它会说明资料结构、关连式资料库模型、程式语言及工具的选用、演算法等等。

当你从头设计一个产品时，最重要的一点就是要确定使用者的体验。要显示什麼样的画面，运作的逻辑为何，要达成什麼功能。再来还要考虑如何达成。如果产品本身要做什麼都还没决定，争论要用哪种程式语言是完全没有意义的。我在这个系列中只会讨论功能规格。

我写了一篇简短的规格范例，应该能让你大致瞭解一份良好功能规格的重点。在我们继续之前请先读这份规格范例。

读完了吗？

一定还没有。现在就去读完再回来，这样我们才能讨论一份良好规格所应具备或不应包含的东西。我会在这裡等你。谢谢。

(耐心地等待...)

噢，很好。你回来了。

下面列出一些我在每份规格上都会放的项目。

一段声明。纯粹自卫用。如果你写一段文字说「这份规格还没完成」，别人就不会衝进你办公室把你的头咬掉。等时间流逝规格开始完整时，你可以改写成「就我所知这份规格已经够完整了，不过有什麼漏掉了，请告诉我」这提醒我每份规格都需要：

一位作者。有些公司认为规格应该要由整组人来写的。如果你尝试过团体写作，就知道那是最可怕的酷刑。团体写作就留给拥有大批新出厂哈佛毕业生的管理顾问公司吧，他们需要做大量的虚工才能收取钜额费用。你的规格应该由一个人负责并撰写。产品规模很大时就切成几区，让不同人写各区的规格。其他公司认为把个人名字放在规格上会让他「独佔功劳」，会太过自我本位，不是良好的「团队合作」模式。胡说。人们对被指派的工作应该要同时有责任和所有权。如果规格有问题，在规格上印上大名的规格负责人应该要负责解决。

脚本。当你在设计产品时，一定要想出某些真实存正的状况以及人们使用的方法。否则最后就会设计出一个完全没有真实用途的产品(就像Cue?Cat)。替你的产品选好客户群，针对每种客户想像一个完全虚构而又完全典型的使用者，用很典型的方式使用产品。我的UI设计书(可以在线上免费取得)中的第9章讨论虚构使用者和情境的建立。这些使用者或情境就是用在这裡的。状况定得愈清楚愈真实，针对真实或虚构使用者的设计就愈好。这也正是我会放入这麼多虚构细节的缘故。

非目标。当你和整组人一起建立产品时，通常每个人都会各有所好，因而产生各式各样真正或虚构的必备心爱功能，如果要将这些功能全部都做出来，恐怕永远做不完而且要花很多很多的钱。所以你必须马上开始删功能，而删功能的最佳方法就是利用规格的「非目标」章节，列出我们不打算做的东西。非目标可能是个不会具备的功能(「没有精神感应式介面！」)，也可能是较一般性的定义(「我们不在意这个版本的效能。这个产品跑得多慢都没关係。如果第二版时间够，我们会把效率最佳化。」)这些非目标很容易引起争议，不过儘早把它们曝露出来却是非常重要的。就像老乔治布希说的：「绝对不会做！」

概要。这就像是规格书的目录。它可以是张简单的流程图，也可以是段广泛的架构讨论。大家会先读这部份知道大致轮廓，再来看细节才有意义。

细节、细节、细节。最后会进到细节。除非必须了解特定的细节，否则大多数人都会略过这一部份。当你在设计一个网路服务时，有个好方法就是把所有可能的画面都取个正式的名称，再对每个画面用一整章钜细靡遗地详述细节。

细节是功能规格中最重要的部份。由范例规格中，你可以注意到我对登入页面各种错误状况有极为详细的说明。当邮件地址错误时要怎麼做？密码错误时要怎麼做？这所有的错误状况，都会对应到将要撰写的真实程式码，不过更重要的是这些状况都会对应到某人必须做的决策。某个人必须决定遗忘密码时的处理方式。由于不决定就无法写程式，所以规格就必须把决定写下来。

未定义项目。第一版的规格有未定义项目是正常的。我在写初版草稿时总是有很多未定义项目，不过我都会标示出来(加上特殊格式便于寻找)，如果可以的话还会讨论各种可选用的方案。等程式人员开始作业时，所有未定义项目都必须处理乾淨。(你可能认为，先让程式员从简单的项目做起，你稍后再把未定义项目定清楚。这是个坏主意。光是处理程式人员实作程式时出现的新问题就够了，根本不会记得还有些旧问题有待处理。另外解决重大项目时所用的方法也可能会严重影响到程式的写法。)

旁注。在写规格时要记住你会有各种不同的读者：程式员、测试人员、行销人员、技术作者等等。当你在撰写时可能会想到一些只对某类读者有用的小资料。举例来说，我会把写给程式员看的讯息(通常描述某些技术实作细节)标成「技术注解」。行销人员直接跳过而程式人员就会细读。通常我的规格裡满满都是「测试注解」，「行销注解」和「文件编写注解」

规格必须是活的。某些程式团队会有一种「瀑布」心态：我们会一次就把整个程式设计好，所以把规格写好印出来丢给程式员就可以收工回家了。对这种想法我只能说：「哈哈哈哈哈哈哈哈！」

这种作法正是规格如此不受欢迎的原因。很多人都对我说：「规格根本没用，因为没有人会照著做。规格总是过时而且从来无法反映产品。」

原谅我。你的规格可能总是过时又不能反映产品。不过我的规格可是时常更新的。只要产品还在开发而又出现新的决策，就会持续进行更新。规格总会反映我们对产品运作的最佳认知。只要在程式完成时(也就是所有功能完备但尚未测试除错)规格才会冻结不变。

为了让大家好过一点，我不会每天重新发行规格。通常我会在伺服器上放一份最新版供大家参考。遇到特别的里程碑时，我会印一份出来，裡面加上修订标记让大家不必重读整份文件(他们可以快速地发现修订标记以便找出变更所在)。

谁该写规格？请看第三篇。

七、轻松写就功能规格说明书 - 第3节：但是……如何？

现在你已经读过为什么要规格和规格里有什么，可以来谈谈什么人该写规格.

谁在写规格？

请容我在此提一点微软的历史.1980年代当微软开始快速成长时，那里每个人都 读过The Mythical Man-Month这本软件管理经典(如果你还没读过这本书，我可 是极度推荐）.这本书的主要重点是说，在进度落后的项目中增加更多程序员， 只会让进度更加落后.这是因为当团体中有n个程序员时，沟通路径的数量会变 成n(n-1)/2,也就是以0(n2)增加.

既然当时众所周知增加程序人员只会让事情更糟，所以微软的程序人员都在思 考要如何撰写更大的程序.

担任微软〃主设计师〃多年的Charles Simonyi提出了主程序员的概念.这个点子 基本上是让一个主程序员负责撰写所有程序，不过他或她底下有整组充当〃程序 奴隶"的菜鸟程序员.主程序员不必费心替每个函数除错，而是定出各个函数的 原型，建立基本结构再丢给菜鸟程序员实作（理所当然的，Simonyi会是最大的 主程序员）.主程序员这字眼有点古老，所以微软用的词是〃程序经理(Program Manager)".

理论上这应该能解决Mythical Man-Month的问题，因为大家不需要互相沟通(各 个菜鸟程序员都只和程序经理沟通），所以沟通路径是以O(n)而非O(n2)成长.

哎呀，Simonyi可能精通匈牙利表示法，不过他不懂Peopleware.没有人会想当 写程序奴隶的.这种系统根本行不通.最后微软发现，尽管有着所谓人月神话 (Mythical Man Month)的问题，还是能在团队中增加聪明人而提升产出，不过边 际价值也会减少.我待在Excel团队时里面有50个程序员，生产力的确比25个人 团队高(不过没有多到两倍）.

主奴式程序作业的点子行不通，不过微软里面还是有一堆叫程序经理的人跑来 跑去.有个叫Jabe Blumenthal的聪明人重新创造了程序经理这个职位.从此之 后程序经理就负责产品的设计及规格.

自此至今，微软的程序经理就是在搜集需求，定义程序应有作用并撰写规格.每 个程序经理通常配5位程序员；程序经理以规格方式定义产品，而程序员则负责 以程序实作写出产品.程序经理也必须负责协调营销，文件撰写，测试，地区 化，以及程序员不会花时间处理的其他烦琐细节.最后当程序员只要全心贯注 让程序正确无误时，微软的程序经理还得心系公司的〃远景".

程序经理是非常宝贵的.如果你曾经抱怨程序员太重视技术优雅而忽略市场性， 你需要程序经理.如果你曾抱怨大家写得一手好程序却写不出好中文，你需要 程序经理.如果你曾抱怨产品方向不明确，你需要程序经理.

你要如何雇到一个程序经理呢？

大部份公司甚至没有程序经理的概念.我觉得这实在是太糟糕了.当我在微软工 作时，公司内程序经理很强的团队都有非常成功的产品（比如Excel，Windows 95, Access).不过其他团队(如MSN 1.0和Windows NT 1.0)则的领导人通常忽视程序 经理角色(这些程序经理反正也不是很优秀，或许本来就该被忽视），而他们的 产品就没那么成功了.

下面列出必须避免的三件事.

1.不要把程序员升为程序经理.良好程序经理所需的技能(撰写清楚的中文，外 交手腕，对市场的察觉，对使用者的认同以及良好的使用接口设计)几乎都不是 良好程序员所需的.当然有人两者都行，不过少之又少.为了奖励好程序员而把 他升到不同的职位(一个改去写中文而非C++的位子），这是个Peter Principle 的典型例子：人们通常会被擢升到他们无法胜任的阶层.

2.别让营销人员当程序经理.这没有不敬的意思，不过我认为我的读者应该会 同意，好的营销人员很少能好好掌握产品设计的技术性问题.

程序管理基本上是完全不同的职业生涯.程序经理一定都是很技术性的，不过 却不必是个好的程序人员.程序经理会研究使用接口接触客户并且撰写规格.他 们必须与各式各样的人为伍，由"白痴"客户到穿着星舰制服上班，孤癖又惹人 厌的程序员，还有身穿2000美元套装大摆架子的销售人员.就某方面而言，程序 经理相当于软件团队的黏着剂.所以领导魅力非常重要.

3.别让程序人员对程序经理报告.这是很微妙的错误.我在微软当程序经理时 设计了Excel的Visual Basic(VBA)策略，并且订出涵盖最微小细节的完整规格， 详细叙述应该如何在Excel里实作VBA.我的规格大约有500页.在Excel 5.0开发 的巅峰时期，我估计每天早上有250人来上班，主要任务就是要实作我写的这份 庞大规格.我完全不知道这些人是谁，不过Visual Basic组就有十几个人光是在 写这玩意的文件(更不用提写Excel的文件或是全职负责说明档内超链接的人了）. 不过最夸张的是我位在这层层结构的"底层没错.没有人要向我报告.如果想 让大家去做某件事，我必须说服他们这件事该做.当主开发师Ben Waldman不想 做我规格定义的某个项目，他跳过不做就好了.当测试人员报怨规格中某个项

目不可能做完整的测试，我就得去把这个项目简化.如果这些人得向我报告， 产品可能不会这么好.因为有些人可能会认为对上司放马后炮不太好.另外我也 可能坚持己见，独断短视地命令他们照我的方式进行.这样做的话就不可能有 机会建立共识.而凝聚共识的决策型式却是做对事的最好方法.

八、轻松写就功能规格说明书 - 第4节：技巧

好吧，我们已经讨论过为什麼要规格，规格裡有什麼，以及谁该写规格。在这系列的第四篇及最后一篇裡，我要和大家分享一份撰写良好规格的建议。

在真正有写规格的团队中最大的抱怨就是「没有人在读」。如果没有人在读，写规格的人通常会变得有点愤世嫉俗。就像以前呆伯特漫画裡，工程师用一大叠4吋厚的规格书堆起来充当隔间。在典型官僚作风的大公司裡，每个人都花好几个月写无聊的规格。等规格写好就放上书架永远不再拿下来，然后重头开始製作产品，完全不管规格裡怎麼写。这是因为没人读规格同时也是因为规格写得无聊透顶。写规格的过程应该是个很好的练习，因为它至少能强迫每个人仔细思考问题。不过规格完成后就被束之高阁(没人看也没人爱)，会让大家觉得这只是件毫无价值的工作。

另外如果没有人读你的规格，等产品完成推出就会出现很多争议。某人(管理阶层，行销人员或是客户)会说：「等一下！你答应过会有蛤蜊蒸锅的！蛤蜊蒸锅在哪裡？」然后程式员会说：「不，事实上如果你看看规格第3章第4节的2.3.0.1，就会看到上面明确写著'没有蛤蜊蒸锅。」不过永远是对的客户可没这麼好打发，所以不爽的程式员只好在产品裡加一个蛤蜊蒸锅(结果让他们更不相信规格)。另外也可能会有个经理说：「嘿，这个对话框裡的文句都太累赘了，而且每个对话框上方应该都有广告才对。」然后程式员会垂头丧气地说：「可是你批准了规格而且裡面明确的列出每个对话框的配置和内容！」不过经理当然不会真正去读规格，因为当他想看的时候就开始心不在焉，而更重要的是这和他週二的高尔夫球比赛有衝突。

所以说虽然规格是很好的，不过如果没人读就不好了。身为撰写规格的人，你必须骗大家来读你的作品，另外也可能要努力不要让大家过份依赖规格而失去原本已经很差的思考力。

要骗大家来读你的作品，通常只需改善写作即可。不过光说句「要当个好作家」就撒手不管实在不太公道。下面列出想让别人读规格时绝对要遵守的四个简易规则。

规则1：要有趣

是的，要骗大家来读规格的第一个规则就是让读规格变得很有趣。别告诉我你生来就没有幽默感。我才不信呢。每个人随时都会有好玩的点子，只不过因为觉得会「不专业」所以自我约束。不过有时候你得要打破规则。

如果你读过我在这个网站写的一堆垃圾，你会注意裡面到处都有想让内容更有趣的拙劣企图。另外往前数第四段我才刚写了个低级玩笑取笑经理在打高尔夫球。虽然我实际上没那麼幽默，我还是很努力在尝试。另外虽说胡扯乱扯试图表现有趣本身就很滑稽，却是属于悲哀的那一型。在写规格时范例是个表达趣味的好地方。当你需要说个故事解释某个功能时，千万不要这样说：

使用者按Ctrl+N建立一个新的雇员表格，并且开始输入雇员姓名。

你应该要这样写：

猪小姐因为手指太肥按不到单一的键，所以拿著一根眼线笔戳键盘按出Ctrl+N建出一个新的男友表格。然后输入单一笔资料"Kermit"。

如果你读过很多Dave Barry的文章，会发现有个很容易的方法可以逗趣，就是在不需要的地方详述细节。「爱吵闹的哈巴狗」就比「狗」有趣。而「猪小姐」也比「使用者」有趣。用「左撇子酪梨农夫」要比「特殊嗜好」好。不要用「不替狗清大便的人应该受罚」，应该说「把他们关在单人囚房，让他们寂寞到想付钱跟蜘蛛上床。」

附带一提，如果你认为逗趣不够专业，那我也没办法，只能说你缺乏幽默感。(不要否认。没有幽默感的人总是不肯承认。你是骗不了我的。)另外如果同事会因为你写的规格轻鬆有趣很好读而轻视你的话，就去找别的工作吧，因为生命实在太短暂，不应该把你的白天浪费在这种苛刻悲惨的地方。

规则**2**：写规格就像在写用脑执行的程式

这就是我会认为程式员写不出好规格的原因。

当你在写程式时，主要对象是编译器。没错，我知道人也会读程式，不过通常会读得很辛苦。对大部份程式员来说，光是要让程式能正常编译就够辛苦了；要把程式写得容易阅读简直是遥不可及。你可以写：

```
void print_count( FILE* a, char * b, int c ){
fprintf(a, "there are %d %s\n", c, b);}

main(){ int n; n =
10; print_count(stdout, "employees", n) /* code
deliberately obfuscated */ }
```

或是

```
printf("there are 10 employees\n");
```

输出是完全一样的。如果你仔细想想，这就是原因所在，程式员通常都会写出如下的东西：

假设有个函数AddressOf(x)，作用是由使用者x转出该使用者的RFC-822格式电子邮件地址，并以ANSI字串输出。让我们假设有使用者A和使用者B，A想送电子邮件给使用者B。所以使用者A用其他任意(不是全部)技术建立新讯息，然后在「传送至：」编辑框内键入AddressOf(B)。

这也可以写成以下的规格：

猪小姐想去吃午饭了，所以打了一封电子邮件并在「传送至：」框内键入Kermit的地址。

技术注解：地址必须是标准的Internet地址(RFC-822格式)

理论上这两种写法的「意思」完全一样，不过除非你很小心的解译，否则是不可能看懂第一种写法的。程式员常会尝试把规格写得像艰涩的学术论文。他们认为一份「正确」的规格必须在「技术」上完全正确，结果完全搅不清楚状况。

问题在于规格不光是正确就好，同时还必须是可理解的，以程式设计的术语来说，就是要把规格写得让人脑能够「编译」。电脑和人脑间最大的差别之一，就是当你定义稍后要用的辞语时电脑会耐心等候。可是对人来说一定要先有动机才会瞭解你在讲什麼。人们不会想被迫解译某些东西，他们只想循序阅读就能理解。对人来说，你必须先提供整体概念然后才填补细节。而电脑程式则是从头到尾顺序下来全部都是细节。电脑不在意变数名称是否有意义。而对人脑而言，如果能先讲个故事(即使只是片断也可以)描绘出鲜明的印象，就能够理解得更清楚。因为我们的脑袋已经演进到能理解故事了。

如果你拿出真实西洋棋赛进行中途的棋局，经验够的老手只要一两秒就能立刻记住每个棋子的位置。不过如果以正常下棋不会走的方法(比如把卒放在第一列或把两隻黑主教都放在黑格上)移动几颗棋子，棋手就很难记下棋局。这和电脑思考的方式不同。对能记忆棋局的电脑程式来说，可能及不可能的棋局记起来都一样。人脑的运作方式并不是随机存取的；我们脑中的某些路径会被持续加强，所以某些常见的事物就会比其他东西更易理解。

所以当你在写规格时得试著想像你的对象是谁，并且想像规格各个部份是要他们瞭解什麼东西。逐句逐句地问自己，就前面已经提供的内容而言，读到这句话的人能深入理解其涵义吗？如果对象中有些人不知道RFC-822是什麼，你必须把定义写出来，或者至少在技术注解中简述RFC-822，这样非技术人员才不会一开始就看到一堆专业术语，结果投降永不再看。

规则3：写得愈简单愈好

千万不要觉得简单句子不够专业，就使用夸大的正式语句。写得愈简单愈好。

人们会认为"use"不够专业所以改用"utilize"之类的单字。(「不够专业」这句话又出现了。不管任何时候，只要有人说你不应该如何如何，因为那样「不够专业」，你就知道这些人已经没有真正的理由了。)事实上我认为很多人会觉得写得清楚明瞭就表示裡面大有文章。

把要讲的东西拆成短句。如果你没办法把某个句子写得很清楚，就把它拆成二或三个较短的句子。避免整面的文字牆：也就是说整页都只有文字。大家遇到这种情况就会吓到根本不敢看。你什麼时候看过有热门杂志或报纸裡整页都是文字的？杂志会从文章裡摘录引言，然后用超大字体印在页面正中央，就是为了避免有整页都是文字的印象。有编号或标上符号的列表，图片，图表，表格以及大量空白，都可以让东西「看起来」更轻鬆。

「杂志会从文章裡摘录引言，然后用超大字体印在页面正中央，就是为了避免有整页都是文字的印象。」

没有东西比撷取大量的画面更能增进规格了。正所谓一图胜千言。撰写视窗软体规格的人都应该买一套Visual Basic，然后学到至少能建立模拟画面。(在麦金塔上就用REAL Basic；写网页就用Front Page或Dreamweaver)。然后撷取这些画面(按Ctrl+PrtSc)再贴进规格内。

规则4：审阅多次并重读几遍

嗯，好吧，本来我打算花大量篇幅来阐述这项规则，不过这一项实在是太简单明显了。把你的规格审阅多次并重读几遍，知道了吗？当你发现某个句子不是非常容易理解，就把整句重写一次。

不去解释规则4省了我很多时间，所以我要额外增加一项规则。

规则5：使用样板是不好的

要避免製作标准规格样板的诱惑。最先你可能只是认为，让「每份规格看起来都一样」是很重要的。我的建议是这并不重要。这会有什麼差别吗？你家裡书架上的每本书都长得一模一样吗？你希望他们一样吗？

更糟的是如果有了样板，很容易就会针对各个自认重要的功能在样板上加上一大堆章节。举例来说，大比尔下令从今以后，所有Microsquish产品都一定要有个Internet元件。所以现在起规格样板裡就有一节「Internet元件」。不管是谁在写也不管内容有多无聊，每个人都得在「Internet元件」那一节填入内容，即使他们只是在写Microsquish键盘的规格。(不然你以为为啥有那麼多无用的Internet购物键像雨后春笋般从键盘上冒出来。)

等这样的章节一多，整份样板就会变得大。(这裡有份非常非常糟糕的规格样板范例。我的天啊，谁会需要在规格裡放参考书目或一份小辞典呢？)这种巨型样板的问题就是让写规格写是很可怕的工作，把大家吓得不敢去写规格。

规格就是一份你希望大家去读的文件。就这种观点来看，规格和纽约客或学校论文没啥不同。你曾听说过有哪个教授会拿样板给学生去写论文吗？你曾读过哪两篇好文章能套进一份样板吗？把这个点子忘了吧

九、轻松制订软件进度表

去年十月，美国东北部充斥着一个名为Acela产品的广告.这是一列由波士顿开 到华盛顿的新开快速火车.到处都是电视广告，大广告牌和海报.你会认为这么 多广告一定创造出一些对Amtrak新快运服务的需求

可能会吧.不过Amtrak根本没有机会知道.Acela—延再延，结果整个营销活动 在Acela服务推出前就结束了.这让我想起某位产品在推出前一个月被媒体大加 赞扬的营销经理，他对我说：”真是大出风头！只可惜你还买不到这玩意！”

那些罣丸素过剩的自大狂游戏公司总喜欢在自家网站上夸称，下个游戏等”好了 “自然就会推出.至于时程?我们才不要那讨厌的东西!我们是超酷的游戏程序员! 大部份公司可没这种福份.问问莲花公司吧.他们的Lotus 123 3.0版初推出时需 要用80286计算机，这在当时可不怎么普遍.结果就把产品延后16个月推出，努 力让程序能在8086的640K内存限制下执行.等他们完成时，微软在Excel的开发 上已经领先了 16个月了，更讽刺的是那时候8086已经完全过时了！

在我写这篇文章时,Netscape的5.0版web浏览器几乎已经拖了两年.部份原因是 他们下了个自杀式的决定，把所有程序丢掉重头开始.同样的错误已经让安信 达(Ashton-Tate),莲花，和苹果的MacOS成为软件历史中的灰烬.Netscape浏览 器的占有率在这段期间由80%左右掉到大约20%,而且完全无法对竞争者有任何 反应，因为他们最重要的软件产品已经被拆成1000片散落在地上，什么事都做 不了.其他因素都不够看，光这一个烂决定就成为Netscape炸烂自己的核弹 了.(Jamie Zawinski’s世界闻名之怒详述了整件事）.

所以说你一定得定时程.几乎没几个程序员想做这件事.以我的经验来说，绝大 部份人都完全不订时程，直接做了才说.至于那少数几个有做的,多是因为老板 一定要，只好敷衍着做一做，而且除了那些同时相信〃软件项目一定会延误〃及” 幽浮”的上级管理阶层外，没人真的相信排出来的时程.

那么为什么没人要订时程呢?主要原因有二.首先是执行起来很痛苦.其次是没人 认为值得做.明明知道排出来不准确，为什么要费事去做呢?大家都为时程一定 是错的，而且时间过得愈久错得愈离谱，所以何必白费工夫呢？

下面提供一个简单无痛的方法，可以订出确实无误的时程.

1)使用微软Excel.不要用像微软Project这种专门的程序.微软Project的问题 在于它假设你要花大量时间处理相关连性(dependency).所谓相关连性是指有 两件工作，第一件工作必须在第二件工作开始前完成.我发现就软件而言，相 关连性实在是很明显，完全不需要花工夫正式追踪.

Project另一个问题是假设你需要能按一个小按钮就〃重新均衡调整 (rebalance)”时程.这表示程序会重新安排把工作指派给不同的人.这就软件来 说完全行不通.程序员是不能互换的.Rita的程序错误交由John来修正的话得花 上好几倍的时间才行.另外如果你把负责用户接口的程序员

拉去处理WinSock问 题，她会卡在那里花整个星期熟悉WinSock程序设计.重点是Project是设计给建筑办公大楼用的，不是写软件用的.

2)简单就好.我用的时程表标准格式简单到你一定背得起来.一开始只要七个字 段就好了： 如果开发人员不只一个，你可以让每个人有自己一个表，也可以加个字段标出 负责各项工作的人员.

3)每个功能应该包含多项工作.所谓功能就像是在程序中增加拼字检查.增加拼 字检查包含好几项程序员必须做的工作.排时程最重要的部份就是排出这个工 作表.接下来是个基本原则：

4)只有实际要写该程序的程序员才能排出该项的时程.任何由管理阶层排好时 程交由程序人员执行的系统一定都会失败.只有实际执行工作的程序员才能找 出完成功能所需的步骤.也只有这个程序员才能估计每个步骤所需的时间.

5)把工作分得很细.要让时程真正产生作用，这可是最重要的一环.分好的工作 应该是以小时计而不是以天计.（当我看到以天甚至以周为单位的时程，我知道 这根本是玩假的）.你可能会认为把工作细分的时程只不过是精确一点而已.错! 错得离谱！当你订时程时先由大项工作开始，然后再细分出细项工作，就会发现 结果完全不一样，而不光是精确一点而已.这样的时程出来的是完全不一样的 数字.为什么会这样呢？

当你必须订出细项工作时，就得强迫自己实际地找出确实要进行的步骤.要编 写foo这个子程序.要建立某些对话框.要读取某某档案.这些步骤都很容易估 计，因为你以前写过子程序建过对话框，也读过某某档案.

如果你随便划分出大项工作(实作出文法修正功能），表示你根本没有真正思考过 要做的事情.而如果你并未真正思考过要做的事情，根本就不可能知道需要多 久才能完成.

依据经验来看，每项工作耗时应在2到16小时之间.如果时程上有项工作需要40 小时(一周），表示细分得还不够.

细分工作还有另一个理由，就是能逼你设计那些要命的功能.如果你排了3周去做 “Internet整合〃这样的粗略功能，你就完蛋了.如果你必须找出要写哪些子程 序，就会逼自己把该项功能弄清楚.因为被迫提早进行这种程度的规划，所以 就能消除软件项目中大量不稳定的成份.

6)记录最初和目前的估计.当你把某件工作第一次排进时程时，先估计所需时 数并填入〇「ig[inal] Est[imate]以及Cu「「[ent] Est[imate]字段中.当计划进行时，你会发现某件工作所需的时间比估计的更长(或更短），就可以依需要更新 Cu「「Est字段.这是由错误学习并教育自己准确估算工时的最佳作法.大部份程 序员并不知道如何估计工时.这是不要紧的.只要你持续学习并随时更新时程， 时程就会确实作用.(你可能必须削减功能或延期，不过时程本身还是能运作无误, 它会一直提醒你该砍功能或是必须延期了）.我发现大部份程序员只要一年工夫 就能排出很好的时程.

当某项工作完成时，Cu「「Est栏的值会和Elapsed栏一样，而Remain栏会变成0.

7)每天更新耗时(elapsed)栏.你不用真的看着马表写程序.只要在回家前或钻 到桌下休息前(如果你是那些疯子之一的话），假装自己已经工作8小时(哈!）并 列出所做的工作，再把8小时分配好填入工作对应的耗时栏中.Excel会自动计算 剩余栏的内容.

同时还要依据现实状况更新各工作的Cu「「Est栏.每天更新时程应该只需要约两 分钟.所以才会说这是无痛的时程排法–快速又容易.

8)加上国定假日,休假等等项目.如果整个时程耗时约一年，每个程序员可能会 休10到15天的假.时程中应该加一项叫〃休假〃的功能，用作国定假日或其他会 消耗人们时间的事情.这样就可以把剩余时间加总起来除以40 (就是考虑一切之 后的所需工作周数），计算出出货的日期.

9)把除错时间排入时程！除错是最难估计的.回想一下你前一个专案.除错所占的 时间很可能是把程序写出来的一到两倍.所以时程中一定要加这一项，而且这 有可能是最大的一项.

实际的作法如下.让我们假设一位开发人员正在做某件工作.〇「ig Est是16小时, 不过到目前为止已经用了 20小时，而且恐怕还要再做10小时.所以开发人员在 Cu「「Est和耗时栏分别输入30及20.

等到达里程碑(milestone)时，这所有的〃落后〃加总起来可能会有相当数量.理 论上为了因应这些延误，我们必须削减功能才能准时上市.幸运的是可以削减 的第一项功能就是名为缓冲(buffer)的功能，而这个项目一开始就排了很多任 务时.

原则上，开发人员会在写程序时除错.程序员在应该除错时绝对不该写新程序. 基于两项原因，随时都应该让错误数目尽可能的少：

1)在写出程序的同一天除错会比较容易.如果一个月后当你忘记程序运作细节 时再来除错，就会变得非常困难而且要花很长的时间.

2)修正错误就像科学活动.不可能估计何时能有发现并解决问题.如果随时都只有 一两个主要的问题，表示未来无法估计的项目不多，所以很容易估算产品推出的时间.反过来说，如果主要的问题有几百几千个，根本就不可能预测什么时候 才能把问题全部修好.

图：

http://chinesetradjoelonsoftware.com/Images/Painless_Software_Schedules.gif

如果开发人员总会在写程序时就把问题修好，为什么还要加上除错项目呢？有 道理，不过即使在写程序时尽量修好所有的问题，在到达里程牌时，测试人员 (内部或外部)总还是会找到真正困难的错误，难免要有许多除错的动作.

10)把整合时间排入时程中.如果你的程序员不只一位，难免会有两人不一致的 事情需要协调.他们会各自建立功能近似的对话框，这当然需要协调.必须有人 细查所有菜单,键盘快捷方式,工具栏工具等等，并且整理及组织所有大家不得不 加的新菜单项.另外只要有两个人把程序登入就会出现编译错误.这也得有人修 正，而且应该列入时程.

11)在时程中加上缓冲时间.事物总是容易用完.你可能要考虑两种重要的缓冲. 第一种:预防工作耗时超过预期的缓冲.第二种:针对未预期但必要的工作的缓 冲(这通常是因为管理阶层决定某功能超级重要,绝对不能等到下一版）.

你可能会很惊讶地发现，休假，国定假日，除错,整合还有缓冲时间加起来超过实际 做事的时间.如果被吓到表示你程序写得还不够久，不是吗？你要忽略这些项目 的话后果自行负责.

12)绝对不要让经理叫程序员缩减估计时间.很多菜鸟软件经理认为能用精细” 紧密(短得不怍实际）”的时程，”激励〃程序人员做得更快.我认为这种激励根本 是脑袋坏掉.当我进度落后时，我会觉得内疚消沈毫不积极.当我进度超前时， 会非常快乐而且充满生产力.时程可不是玩心理游戏的地方.

如果你的经理要求你缩短估计时间，这里告诉你要怎么做.在时程表上加一个 叫Rick的估计（当然是假设你叫Rick)的新字段.把你的估计填进去.随便经理怎 样要求，直接把她定的时间填入Cu「「Est字段后就不要管了.等项目经束时再看 看谁的估计比较接近实际状况.我发现光是威胁说要这样做，效果就很惊人了， 特别是当你的经理了解到，他们刚参加了一个看你能做得多慢的竞赛时更是有 效！

为什么不适任的经理们总会试图要程序员缩短估计时间呢？

当项目开始时技术经理会去见经营人员，然后会得出一个他们认为三个月（实际 上要9个月）做得到的功能列表.如果你认为写程序不需要先想清楚所需步骤， 然后估算出来某工作需时n，实际上很可能会耗时超过3n.在订定真正的时程时， 把所有工作加总起来，就会了解项目耗时远比想象中多得多.欢迎光临真实世界.

不适任的经理的处理方法是想办法让员工做得更快.这一点其实不太实际.你或 许能雇用更多员工，不过他们需要时间适应，可能前几个月都只有一半的效率 (还会拖慢必须引导他们的其他人员工）.而且无论如何，在这个业界得要6个月 才找得到好的程序员.

你可能可以让员工在一年内全力以赴，暂时提高10%的初版程序(译注:指未整理 除错的程序)产量.算不上是什么大跃进，而且这样有点太短视了.

你可能可以恳求员工不计辛劳超努力地工作，提高20%的初版程序产量.砰!可惜 除错时间倍增了.真是了不起的自爆蠢方法.

不过你绝对绝对不可能由n变成3n，如果你自认有这种本事，请写信告知贵公司 的股票代码好让我放空.

13)时程就像积木.如果你有一堆积木，积木太多塞不进箱子里.这时候你只有 两个选择：找个大点的箱子或拿掉一些积木.如果你认为能在6个月内完成出货, 可是时程上排的却是12个月，同样的也只能延后出货时间或是删掉部份功能. 积木是不能压缩的，如果你自认为可以，那只不过是在骗自己，徒然让自己失 去一个真正能展望未来的好机会.

另外要知道，如此维护时程还有其他好处，就是能逼你自己删除功能.为什么 说这是好处呢？假设你有两个功能：其中一个非常有用而且能让产品变得超棒 (比如Netscape 2.0里的table)，另一个很容易而且程序员很想写(如BLINK标 签），不过却啥用处也不具备市场价值.

如果你不订出时程，程序员会先做简单/有趣的功能.然后等时间用完时你完全 没得选择，只能延后时程来完成有用/重要的功能.

如果你在开始作业前就排出时程，就会了解必须削减某些项目,自然会把容易/ 有趣的功能砍掉而做有用/重要的功能.这样子强迫自己削减某些功能，就能完 成更强更好的产品，不但功能更好而且又更早推出.

我想起Excel 5的制作.我们最初的功能列表非常庞大，远远超出我们的时程. 我们都在想:天啊！这些全都是超级重要的功能!没有宏编辑精灵我们怎么活得 下去呢？

结果是我们没得选择，所以只好配合时程把功能删到不能再删.每个人对于削减 功能都很不爽.为了安抚自己的感觉，我们只好告诉自已说这不是在删除功能， 只不过因为这些功能没那么重要，所以延后到Excel 6而已.

当Excel 5几近完成时，我和同事Eric Michelman开始写Excel 6的规格.我们坐 下来审视由Excel 5删除移过来的"Excel 6"功能列表，当我们发现这些被删除的 功能烂极了时真的是被吓到了.里面竟然没有一项值得做.我认为即使是接下来

的三个版本，也完全没有必要制作这些功能.配合时程削减功能是我们所做过 最好的事.如果我们没有这样做，Excel 5可能会耗双倍时间而且包含50%无用 的垃圾功能.(我绝不怀疑这就是Netscape 5/Mozilla现在的状况:他们没有时程 也没有明确的功能列表，没有人愿意削减功能，所以永远无法推出.等他们能 推出时，里面会有很多IRC客户端等根本不该做的无用功能.

附录：你该知道的Excel二三事

用Excel管理软件时程如此好用，原因之一就是对大部份Excel程序员而言，

Excel的唯一用途就是维护软件时程表！（只有少数程序员会用沙盘推演 (what-if scenario)规划的方式做事的…这些程序员！

共享列表利用档案选单的共享列表命令可以让大家同时开启并编辑档案.由于 整个团体应该会持续更新时程，这个功能帮助很大.

自动筛选这是个筛选时程的好方法，举例来说，你可以只检视所有指派给你的 功能.再加上自动排序功能，就能依优先度顺序检视所有指派给你的功能，这 实际上就是你的工作清单.够酷吧！

枢钮分析表这是个检视摘要及多重汇总表格的好方法.举例来说，你可以做一 张图显示各优先度下各个开发人员的剩余时数.枢钮分析表就实在是个大创新. 你一定要学会怎么用枢钮分析表，因为它能让Excel增强一百万倍.

Excel分析工具箱中的WORKDAY函数是个在无痛时程中计算日历天数的好方法.

十、每日连编是朋友

自动地一因为你设定代码每天在固定的时间构建。在Unix环境下使用cron在windows下使用“任务 计划”。

每天一或者更频繁.当然每天构建的次数越多越好啦。但是有时候构建次数还是有上限的，原因和 版本控制有关系，等会儿我会谈到的。

完整地一很可能你的代码有多个版本。多语言版本，多操作系统版本，或者高端低端版本。每日构建 (daily build)需要构建所有这些版本。并且每个文件都需要从头编译，而不是使用编译器的不完美的增 量编译功能。

以下是每日构建（daily build)能带来的好处：

当一个bug被修正了，测试者可以很快得到最新的修正后的版本开始重新测试，以验证bug是否真正地 被修复了。

开发人员可以更加确定他们对代码做的修改不会破坏1024个操作系统上的任何一个版本。验证这一点 不需要在他们的机器上安装OS/2 (IBM公司生产的PC机操作系统）。

那些每天将修改过的代码导入（check in)版本控制服务器的开发人员知道，他们对一个模块导入的 修改不会拖别的开发人员的后腿。拖后腿的意思是，那些开发别的模块的程序员使用这个修改过的模块，

出了问题，于是他们自己的模块也没有办法开发下去了。每日构建则不会有人拖后腿。如果把一个开发团 队比作一台PC机，那么团队中的一个程序员对某个模块的修改导致其他人无法开发别的模块，相当于PC机 发生了蓝屏。当一个程序员忘记把他（她）新建立的文件导入到repository (指版本控制服务器上的代码 树）时，这种开发过程中的“蓝屏”会经常发生。因为在这个程序员自己的计算机上有这个文件，所以他 (她）构建这个程序没有问题。但是其他程序员是从版本控制服务器上导出（check out)代码的，由于 服务器上没有这个文件，他们遇到了链接错误（link e「「o「），无法继续工作了。

外部团队（例如市场销售部门，进行beta测试的一些客户）可以获得一个比较稳定的版本，这样对他 们开展自己的工作比较有利。

假如你将每日构建出的二进制文件（例如一个可执行程序，一个dll等等）存档管理，那么当你发现一 个非常奇怪的，无法解决的bug时，你可以通过对这些文件进行二进制搜索（binary search)来确定什么 时候这个bug第一次出现。如果有对代码进行了完善的版本控制，你也可以找出是谁在何时对代码进行的导 入（check in)导致了这个bug。

当开发者修正了测试者报告的一个错误时，如果测试者同时报告了发现错误时的构建的版本，开发人 员可以直接在那个版本中测试是否bug真正被修复了。（译者按：测试者报告出现了一个bug,只是报告了 一个错误症状，而错误的原因可能有多个，每个原因可能在不同的

模块中。前文中的方法是为了避免只修

正了一个模块中一个原因，别的模块由于在变动，于是掩盖了而不是修复了bug)以下是如何进行每日构建（dai lybui ld)的具体步骤。你需要用最快的电脑建立一个每日构建服务器。 写一个脚本，可以自动从版本控制服务器中导出（checkout)完整的代码，（你当然使用版本控制，不是 吗？），然后对代码从头开始进行构建（build)，要构建所有的版本。如果你有一个安装打包程序，也要 在脚本中自动运行。所有会卖给最终用户的东西都要包括在构建过程中。把构建出来的版本放在各自的的 目录里，不同时间构建的相同版本也应该按日期整理好，不要相互覆盖。每天固定的时间运行这样的脚本。

最关键的是所有这些步骤都应该由脚本自动化完成，从导出（checkout)代码到将最终产品放在网上 供用户下载（当然在开发阶段，产品是放在一台测试服务器上的）。要保证开发过程中的任何东西的任何 记录是由文档记录的而不是由某个人的脑子来记录的，这是唯一的办法。否则你会碰到这样的情况，产品 需要发布了，可是只有Shaniqua知道如何将产品打包的，可是他刚刚被巴士撞了。在Juno公司（本文作者 工作过的公司之一），要进行每日构建，你唯一需要学会的东西就是双击每日构建服务器桌面上的一个快 捷方式。

如果你在发行程序的当天发现了一个小bug,没有问题。修正它，然后重新运行每日构建脚本，现在你 可以安安心心的发行程序了。当然，黄金定律是每日构建脚本应该是把所有的事情从头做一遍，遵循这个 定律就不会有什么问题。

将你的编译器的警告参数设到最大（在微软的VC中是-W4 ;在GCC中是-Wall),当遇到任何一个最微 小的警告时就应该停下来。

如果每日构建失败了，可能整个开发团队的工作会因此进行不下去。当务之急是立刻找出原因，使得 每日构建能成功进行下去。某天也许你会一天运行好几次每日构建脚本的。

每日构建一旦失败，应该自动地将失败用email通知整个团队。提取错误日志中的恰当部分并包括在 email正文中也不是很难。每日构建脚本也可以将当前的状态报告整理成一个html文件，自动发布到一个所 有人都可以访问的web服务器上，这样测试者可以很快知道那个版本的构建是成功的。

当我在微软的excel团队中工作时，我们的一个有效办法是，谁导致每日构建（daily build)失败， 他（她）就得负责照看当日的每日构建（译者按：微软通常每日构建都在半夜），直到有另一个人导致构 建失败而接替他（她）。这样做当然可以使每个开发者都小心不要因为自己代码的错误破坏了构建，更重 要的是团队中的每个人都可以学会每日构建（daily build)的原理。

如果你的团队在同一个时区工作，在午饭时间进行每日构建（daily build)是个不错的主意。午饭前 每个程序员导入（check in)代码，这样当程序员在吃饭时，构建系统在工作。当程序员吃完饭回来时， 如果每日构建失败了，所有的人也都在，可以尽快找出失败的原因。当构建系统运作起来时，没有人再会 担心别人导入（check in)代码会妨碍自己的工作了。

如果你的团队同时在两个时区工作，计划好每日构建（dailybuild)的时间使得一个时区的工作不会 影响另一个时区的工作。在Juno公司，纽约程序员在晚上7: 00导入（checkin)到版本控制服务器。如果 他们的导入导致构建失败，印度Hyderabad市（译者按：印度科技重镇）的程序员在纽约时间晚上8: 00以 后的工作几乎无法进行下去。我们每天进行两次每日构建（dailybuild)，每次构建的时间都在两地的程 序员回家之前，这下就没有问题了。

十一、难伺候的故障修复

TRS-80 Level-I BASIC只能储存两个字符串变量姊和郎.同样的我脑袋里天生就 只够放两只程序臭虫(bug).不管何时我只能记住两只臭虫.如果你要我记住第 三只，先前的某一只，就会自然而然的被遗忘在荒烟漫草的回忆中。

拥有记录问题的数据库是优秀软件团队的标记之一.事实上只有极少数团队有 实际进行，这一点一直令我很惊讶.程序人员似乎全都自认能用脑袋或立可贴 记住所有错误，这件事实在错得离谱.

如果你能专心听一下，我想要延续我前几篇文章无痛时程表和无痛规格的精神, 说明一种能无痛地进行错误追踪的方法.

首先你需要一个真正的数据库.如果是两人一组在周末程序课上写些小程序， 拿文本文件当数据库大概还可以.不过只要规模再大一点就要用真正的错误追 踪数据库.市面上有无数的错误追踪数据库随你选买.（厚脸皮自我推销一下：

我们在Fog Creek Software也写了一套，名为FogBUGZ.要我来形容的话，这是个功能强大而且使用简单的网页型产品.）

为了示范作用，让我们跟着臭虫走走，看着它从出生一直到某人让它解脱为止 的生活.我们将要跟着大名鼎鼎的1203号臭虫.下面是错误数据库对这只虫的叙述：

```
1203
Bee Flogger 2.0 FTP客户端
上传档案会导致FTP服务器倾印核心(dump core)
已关闭
结束(已解决-已修正)
2 -必须修正
2.0 Alpha Build 2019
Jill 的 iMac, Mac OS 9.0, 128M RAM, 1024x768 millions of colors
11/1/2000由超级测试员Jill发现
* 启动Bee Flogger
*建立一个未命名档案，内容只有一个"a"字 *点工具栏上的FTP钮 *尝试用ftp传至你的服务器
错误：观察；ftp服务器停止响应.输入ps-augx显示ftp服务器 甚至停止执行而且根目录有核心倾印.
预期：不应当掉
11/1 /2000由Jill超级测试员指派给指派给开发主管Willie
11/2/2000 (昨天) 已解决-开发主管不予修正
不是我们的程序, Jill,那只是Linux所附的proftpd.
11/2/2000 (昨天) 已由超级测试员Jill重新启动(指派给开 发主管Willie)
这不对劲.我用一般ftp客户端联机都不会让proftpd当掉.可是 用我们的程序每次都会当.Ftp服务器不会自己"当"掉.
11/3/2000 (今天) 已由开发主管Willie指派给程序人员 Mikey
,你能不能看一下这个问题？你的客户端程序可能有毛病.
11/3/2000 (Today)已解决-已由程序人员Mikey修正
我想我把密码错传成用户名称或其他东西了...
11/3/2000 (今天) 已由超级测试员Jill重新启动(指派给程 序人员Mikey)
Build 2021还是有这个问题.
11/3/2000 (今天) 已由程序人员Mikey编辑
呃.这蛮奇怪的.让我来抓这个错.
11/3/2000 (今天) 已由程序人员Mikey编辑
我想应该是MikeyStrCpy()的问题...
11/3/2000 (今天) 已解决-已由程序人员Mikey修正
啊！
终于修好了丨
11/3/2000 (今天) 已由超级测试员Jill关闭
build 2022显然修好了，所以我把这个问题关闭了.
```

以下是实际发生的事情.

程序员Mikey正为他的麦金塔软件写新的FTP用户功能.写到一半时由于觉得不 爽，就写了自己的字符串复制函数.想藉此教训教训公司里那些强迫要重复使用 程序代码的家伙!哇哈哈！

Mikey,当你不重新使用程序代码时就会出事.而现在所出的事就是Mikey忘记把复制好的字符串加零字符作结尾.不过他一直都没有注意到这个问题，因为 大部份情况下，刚好都是把字符串复制到已先清成零的内存里.

同一周稍后超级测试员Jiii正在猛操那个程序，她把额头顶在键盘上滚来滚去 或进行某些同等严苛的测试.（恰巧的是大多数优秀的测试员都叫做Jill或 Gillian等等.）突然间发生了非常奇怪的事情：她测试用的ftp服务器当掉了. 不要怀疑，我知道这是台Linux机器而Linux机器是不会当的(slashdot的人拜托 不要嘘我）.不过这个该死的东西就是当了.而她根本没有动过服务器，她只不 过用Mikey的Mac程序把档案FTP上去而已.

现在呢，由于Jill是个超级优秀的测试员，所以她会小心地把所做过的事记下 来了（比如在实验手册上精确记录头在键盘上滚动的方位）.她找一台干净的机 器从头开始重复每个步骤，看吧，又发生了！Linux的ftp daemon又当了！这么 一来一天内就发生两次了！ Linus(译注:创造Linux的人)注意啦.

Jill斜眼看看重现步骤清单.总共大约有20个步骤.其中有几个似乎与问题无关. 做了一点实验后，Jill已经能把问题缩减到四个步骤，而且每次都能产生相同 的结果.现在她已经准备好把问题发出来了.

Jill在问题追踪数据库中输入新错误.这里光是输入错误的动作就是有学问的： 有好的错误报告也有不好的错误报告.

优良错误报告必备的三元素

> 然后主说话了，祂说"你要先取出圣撞针.然后你要数到三，不多，不少.三是你要数的数而要数 的数就是三.四不是要数的数，二也不是,除非接下来要数到三.五完成不行.当数到第三个数字 三时，就把你的安提阿金手榴弹投向你的敌人，那些在我眼中极其下流的人，他们必须承受" > –圣杯传奇

写一份优良错误报告的规则很好记.每个好的错误报告都要有刚好三个东西.

1. 重现的步骤，
2. 期望看到的，以及
3. 实际所看到的.

看起来很简单，对不对？可能没那么容易，身为程序员，别人丢给我的问题总 会缺一两点.

如果你没告诉我要怎样重现问题，我可能根本不知道你讲的是什么."程序当了 而且在桌面留下一个臭臭的大便状物体."讲得很好，宝贝.不过除非你告诉我 你做了些什么，否则我是什么事都不会做的.现在我得承认有两种情况是很难 找出重现步骤的.有时候是根本不记得或只是在转述〃田野〃中(field，译注:指 实验室以外的环境）的错误.(顺便提一下，为什么他们要称之为〃田野〃呢？是不 是像黑麦田或其他作物呢？不管了…)还有另一个可以不提供重现步骤的场合， 就是问题时有时无并非一直出现，不过你还是应该提供重现步骤，再加个小附 注标明问题不是时常出现.在这些场合下要找到问题实在是很难，不过我们还是可以试试.

如果你不指明你期望看到的，我可能不明白它为什么是个问题.开机画面上有 血迹.那又怎样？我写这段程序时割到手指啦.你希望看到什么？啊，你说规格 上说没有血迹！现在我搞懂你为什么说它是个问题了.

第三部份.你实际上看到什么.如果你不告诉我这一点，我不会知道问题是什么. 这是理所当然的.

回到我们的故事

Anyhoo. Jill把错误输进走了.在良好的错误追踪系统中，错误会自动分派给该 项目的开发主管.这里隐藏了第二个概念：每个错误随时都要有一个人负 责， 一直到错误关闭为止.错误就像是个烫手山芋：当拿到烫手山芋时，你就得负 责把它解决掉，或是把它丢给别人.

程序主管Willie看看这个错误，认为这个可能与ftp服务器有关，所以就以"不 予修正〃把它处理掉.毕竟他们并没有写那个ftp服务器.

问题被处理掉之后就会分派回开启的人身上.这可是个重点.不是程序人员认为 没事就真的没事了.铁律是只有开启错误的人才能关闭错误.程序人员可以解决 问题，意思是"嗨，我觉得弄好了"，不过要实际关闭错误并把它从清单中拿掉， 最初开启的人必须确认问题真的已被修好，或是同意该问题基于某于原因不必 修正.

Jill收到一封电子邮件通知她问题回来了.她看看信里开发主管Willie的说明. 有些东西不太对劲.这套ftp服务器大家用好几年了都没有当.只有用Mikey的程 序才会当.所以Jill重新启动该问题并说明她的想法，所以问题又回到Willie 身上.这次Willie把问题指派给Mikey修正.

Mikey看看这个错误，认真的想了很久，结果完全误诊了这个问题.他修正了其 他完全不相干的错，然后就说把Jill开启的问题解决掉了.

问题回到Jill身上，这次标示为〃已解决-已修正Jill拿最新版软件试了她 的重现步骤，看着看着Linux服务器就当了.她又重新开启问题并且直接指派回给Mikey.

Mikey被难到了，不过最后还是找到了错误的根源.（知道是什么了吗？我要把它 当作给读者的作业.线索可是给足了！）他把问题修好后测一测，然后–找到了！ 照重现步骤去做不会再让ftp服务器当掉了.他再一次标示〃已修正〃把问题解决 掉.Jill也试了重现步骤，发现问题的确修好了，所以就把它关掉.

错误追踪的十大建议

1. 好的测试员一定会试着把重现步骤减到最少步；这对必须追出问题的程 序人员来说极为有用.
2. 要记住只有一开始开启的人才能关闭该错误.其他人可以解决问题，不 过只要最初看到错误的人可以真正确认所看到的错误已被修正.
3. 要解决错误有很多种方法.FogBUGZ中可用的解决方法有fixed(已修正）， won't fix(不予修正），postponed(暂缓），not「ep「o(无法重现）duplicate(重复的问题），〇「by design(设计限制）.
4. Not Repro表示没有人能重现该错误.程序人员在错误报告漏写重现步骤 时常常会用这一项.
5. 你需要小心追踪程序版本.每次发给测试员的软件都应该有个编译ID编 号，这样可怜的测试员才不必在根本尚未修正的版本上测试同一个问题.
6. 如果你是个程序员，而且想尽办法都无法让测试员使用问题数据库，只 要不接受用其他方法写的错误报告就可以了.如果测试员习惯用电子邮件 送错误报告给你，你就把信退回去并加上以下简短讯息："请把它放进问 题数据库中.我没法子追踪电子邮件
7. 如果你是个测试员，而且想尽办法都无法让程序人员使用问题数据库， 只要停止向他们报告错误就好了-把错误直接放在数据库里并让数据 库发信通知.
8. 如果你是个程序员，而且只有部份同事在用问题数据库，只要开始在数 据库里把错误分

派给他们.最后他们自然会明白的.

9. 如果你是个经理，而且似乎没人在用你花大笔费用安装的问题数据库，那就开始利用它把新功能分派给大家.因为问题数据库也是个很好的"未 完成功能〃数据库.
10. 要避开在问题数据库里增加新字段的诱惑.每个月总会有人想出个要在 数据库里新增字段的好点子.什么聪明的点子都有，比如追踪发现问题 的档案；追踪有多少比例的问题是可重现的；追踪某错误发生的次数； 追踪某问题发生时机器上某个DLL的版本.不要屈服在这些点子之下，这 一点非常重要.如果你屈服了，输入新错误的画面最后会出现上千个字段要输入，结果是没有人会想输入错误报告.要让问题数据库有效运作， 一定得让大家都用，如果〃正式〃输入错误太麻烦，大家就会绕过错误数 据库.

只要你在写程序（只有一个人写也一样），如果没有一套良好的数据库列出程序 中所有的问题，一定会产生质量低劣的程序代码.良好的软件团队不只会广泛 使用问题数据库，其中成员还会习惯用问题数据库产生待办事项列表，并把浏 览器首页设成会列出所分派到的工作，还会开始祈祷他们能把问题指派给办公 室经理，叫他进多点山露汽水(Mountain Dew，译注:百事可乐的著名饮料）.

Fog Creek Software制作了 一套易用的问题追踪软件，名为FogBUGZ.

十二、软件开发中的5个世界

有一件重要的事几乎在所有程序设计和软件开发文献里都没提到，所以我们之间有时会有误解。 你是个软件开发者，我也是。不过我们的目标和需要可能并不相同。事实上软件开发有几个 不同的世界，而且每个世界的规则都不一样。 去找一本关于UML模型的书来看看，里头完全不会提到UML不适合用于驱动程序。或者你去读 另一篇说「20MB执行时期链接库[.NET要用的]并不是问题”的文章，它也并没有指出明显的 事实：如果在只有32KB ROM的呼叫器上写程序，这绝对是个问题！

我认为这里头有五个世界，虽然偶而有重迭但通常都不会。这五个分别是：

1. 热缩封膜(Shrinkwrap)软件
2. 内部用的软件
3. 嵌入式软件
4. 游戏软件
5. 用后即丢的软件

当你读最新的极限程序设计书藉或Steve McConnell最好的书，或是Joel on Software或软 件开发杂志，会看到很多有关如何开发软件的声明，不过几乎都不会提到所讨论的开发类型，这实在是很不恰当，因为有时候世界不同，做事的方法也会跟着变。

让我简短的介绍这些类型。

热缩封膜软件是「外头」很多很多人要用的软件。可能是用玻璃纸封好在CompUSA贩卖，也可能是透过Internet下载的。可以是商业软件或共享件，也可以是开放源码或GNU或其他型式-重点是该软件会被成千甚至数百万人安装使用。

热缩封膜软件由于两个特性而衍生出特有的问题：

1. 由于使用者很多而且通常都有代替商品，所以用户接口必须比一般水平更容易才会成功。
2. 由于软件会在很多的计算机上执行，所以程序对计算机间的差异要格外有弹性。 上星期某人寄了一封电邮给我，内容是CityDesk—个只在波兰版Windows出现的问题，原因 是那个操作系统会用右边的ALT键输入特殊字符。我们测过Windows 95、95OSR2、98、98SE、Me、NT 4.0、Win 2000还有Win XP。也测过安装了 IE 5.01、5.5或6.0的状况。我们也测了 美国、西班牙、法国、希伯来文、和中文各种语文的Windows。可是就是还没测到波兰版。

热缩封膜软件有三个主要的分支。开放源码软件通常是不支薪的人开发的，而这些人的动力经常会剧烈变动。举例来说，在全都是志愿者的团体中，被认为不「好玩」的事通常没人会做。另外也如Matthew Thomas所指出会降低可使用性。开发动作很可能是分散在各地进行，

导致团队沟通的质量参差不齐。在开放源码世界很少会面对面在白板旁一边画着框框和箭头一边讨论，所以这类项目在需要画图沟通的设计决策上通常都做得很差。因此这种地域分散团队在复制现有软件时做得会好很多，因为抄现有的软件就不太需要设计了。

顾问软件是另一种热缩封膜软件分支，由于需要很大量的客制化和安装，所以得用恐怖的价钱请一整批顾问来安装。客户关系管理(CRM)和内容管理系统(CMS)软件通常都属于这一类。有些人会觉得这些软件其实什么事都做不了，只不过是找一群顾问进来每小时付300美元的借口罢了。虽然顾问软件装得像热缩封膜软件，不过由于安装启用的成本太高，其实比较像内部用的软件。

像Salesforce.com或eBay之类的Web商业软件还是必须容易使用并能在很多浏览器上执行。虽然开发者比较幸运，能或多或少控制所部署的环境(位于数据中心的计算机)，不过还是得处理各种浏览器间广泛的差异，以及大量的使用者。因此我认为它基本上算是热缩封膜软件的一支。

内部用软件只考虑一种状况，在一家公司的计算机能跑就好了，因此开发起来容易多了。你可以对执行环境做各种的假设。你可以要求某个版本的Internet Explorer或微软Office或Windows。如果你需要幽图表，呼叫Excel来幽就好了；我们部门里每个人都有Excel。（不过 在热缩封膜软件这样做的话，潜在客户大概会跑掉一半。）

在这里可使用度并不重要，因为必须使用的人数有限，反正也没有其他选择就只能将就。开发速度会更加重要。由于开发投入的价值只局限于一家公司，能用的资源相对上也少了许多。 微软花得起五亿美元开发一套对一般人只值80美元的操作系统。不过当Detroit Edison(译 注：能源公司）开发一个能源交易平台时，投入的成本就必须适合单一家公司。要得到合理 的投资报酬率就不能像热缩封膜软件一样花钱。所以很悲哀的，很多内部用软件都烂到不能 再烂。 嵌入式软件具备一个特性，它会被放在硬件里而且几乎都不能更新。这可是个完全不同的世 界。由于没有第二次机会，质量要求比平常高出许多。你可能得用一颗比一般桌上型处理器 慢上许多处理器，所以得花很多时间做优化。程序漂不漂亮没关系，跑得快才重要。可用的 输入及输出装置可能也很少。 上周租的车上有装全球定位系统(GPS)，不过上面的输入/输出很差，几乎不能用。你曾试过 用这种玩意输入地址吗？他们在屏幕上画了一个「键盘」，你必须用箭头键从五个小矩阵(每 个里面各有9个字母)选出要的字母。（我自己车上的GPS有触控屏幕，使用接口就好很多多。 不过离题了。）

把游戏软件独立算一类有两个原因。首先游戏开发的经济是打击导向的。有些游戏击出安打， 更多的游戏被三振，如果你想在游戏业赚钱就得了解这一点，然后确定你有多个游戏，才能 用大赚的钱去平衡失败的损失。这其实比较像电影业而不是软件业。

游戏开发更大的问题是只能有一个版本。当使用者玩完毁灭公爵3D版本，并不会因为修正了某些问题或增加新武器而升级成毁灭公爵3.1D。除了少数例外，一般玩完一个游戏后再重玩是很无聊的。所以游戏的质量要求和嵌入式软件相当，又要有足够的钱一次就把东西做好。热缩封膜软件开发者就幸福多了，如果1.0版不符合大家的需求，再出个2.0版或许就可以了。

最后用后即丢软件是只为了得到其他东西而暂时创造的软件，当你达到目的之后永远都不会再用到。举例来说，你可能会为了其他需要，写一个小命令脚本把某个输入档案转成需要的格式，这就是只用一次的操作。

或许还有其他我忘记的软件开发类型。

这里要讲一件重要的事情。当你阅读那些由专业软件开发大师或顾问撰写的程序设计方法书时，可以放心认定书里说的都是企业内部用的软件开发。不是热缩封膜软件，不是嵌入式软件当然也不会是游戏软件。为什么呢？因为这些大师都是企业在请的，是企业在付他们的薪水。（相信我吧，id software是不会请Ed Yourdon去教结构化分析的。）

上星期Kent Beck声称如果实施极限程序设计的话，就不需要错误追踪系统。因为成对程序 设计(持续进行程序代码检视)和测试导向开发(保证100%的自动测试涵盖率)的结合，让你几 乎不会出现问题。这我可就听不懂了。于是就到我们自己Fog Creek的错误追踪数据库里看 看究竟都是在忙些什么问题。

瞧，我发现这些问题绝少能被成对程序设计或测试导向开发找出。我们很多的「问题」基本上就是XP所谓的story，只是功能要求。我们用问题追踪数据库来记录要实作的大小功能和 改善，并且用它来管理这些工作并排定优先级。

其他很多问题都是只会在外头经过大量使用后再会发现。比如波兰键盘的问题。这种东西成对程序设计是不可能发现的。还有那么我们未曾发生，把不同功能一起运作时才出现的逻辑错误。程序愈大愈复杂，功能间的未考虑到的互动就会愈多。一个很少出现的特别字符序列(`{${?` ，如果你一定要知道的话)让字汇分析器错乱了。有些ftp服务器在删除不存在的档案 时会出错(我们的ftp服务器不会有事，所以我们从来没遇到过。)

我很小心的看过每一个问题。在CityDesk更新版所修正的106个问题中，只有5个可以藉由搭档程序设计或测试导向设计来预防。事实上我们知道而且觉得没关系(这只能让我们的客户 来指正！)的问题远比XP方法能抓到的要多。

不过就其他开发类型来说Kent是对的。就大部份的企业开发应用而言，上面这些东西都不算问题。输入不合法时程序会当掉？重新执行一次，这次注意不要输入{${?了！另外我们只有一种FTP服务器而且全公司都没有人在用波兰版的Windows。

不管在做什么项目，大部份软件开发的事情都是一样的，不过并不是每件事都是。当某 人告诉你某个方法并认为可以用于你正在做的工作时。先想想这个家伙是打哪来的。Steve McConnell、SteveMaguire还有我都来自相同的小角落：一个在华盛顿州雷蒙撰写客户众多的电子表格应用程序的世界。因此我们对容易使用有很高的要求，同时要求问题的数目要少。 其他方法论大师的工作是企业内部开发的顾问，所以他们会说那些话。不管如何，我们都应 该能由对方身上学到一些东西。

十三、稿纸原型开发

几年前Excel团队想知道，让用户能用鼠标拖放电子表格格子好不好。他们叫几个实习 生用最先进的Visual Basic 1.0「快快地弄出一个原型」以便作可用度测试。结果却花了整 个夏天，因为这个原型必须重现很多真正的Exce l功能，否则就没办法做真正的可用度测试。

可用度测试的结论呢？没错，这是个好功能！负责的程序员只用了大约一星期就实作出 完整的拖放功能。这个笑话当然就是原型的唯一意义就是要「节省时间」。

一年后另一个最高机密的微软团队又用最先进的产品Asymetrix Toolbook(天啊，真不 敢相信这东西还活着），替一套新使用接口制作完整的原型。原型花了大约一年才做好。而 真正的制品Microsoft Bob,(软件界的PCjr)的下场是另一套废弃的原型。

基本上我已经放弃软件原型了。如果原型可以执行所有产品的功能，倒不如直接当作产 品，如果不能的话原型也没多大用处。还好有个更好的点子：纸上原型。它一下子就彻底的 解决了这个问题和冰山问题。更幸运的是，Carolyn Snyde「刚刚针对这个主题写了一本很好 的新书：纸上原型制作。这本书对任何设计使用接口的人来说都是很基本的参考数据，而且 书写得很好。

十四、不要被太空架构师所吓倒

伟大的思想家想问题时会开始看出模式。他们看到人们要传送字处理档案给别人，又看到人们要传送电子表格给别人，然后就会发现里面有一个通用的模式：传送档案。这已经是一层的抽象。然后他们会再上一层：人们传送档案，不过网页浏览器也会「传送」网页要求啊。另外如果你有想到，其实呼叫某个对象的方法也像是传送讯息给该对象!这又是一样的东西！这些全都是传送的动作，所以我们聪明的思想家就发明了一种更新更广义的新抽象概念：「讯 息传送(messaging)」，不过现在这件事变得愈来愈神秘，再也没人真的懂他们在说些什么。 废话。

当你一直往上把事情弄得太抽象，就会像上太空一样没有氧气。有时候这些聪明的思想家就是停不下来，然后就创造出这些荒唐又无所不包的高层次宇宙景像，这些东西什么都好，就是完全没有实际的意义。

这种人我称之为架构航天员。要他们写程序或设计程序是难上更难，因为他们没法子不想架构。叫航天员是因为他们活在氧气层之上，我不知道这些人是怎么呼吸的。他们通常在真正的大公司上班，只要这种公司才养得起大批不事生产，完全没有贡献的高学历份子。

最近有个例子可以拿来说明。典型的架构航天员会针对「Napste「是个用于音乐下载的点对点服务」这件事，把架构以外的东西全部忽略掉，然后说因为是点对点所以很有趣，完全没搞清楚重点在于可以输入歌名后可以马上听到。

他们就只会说点对点如此这般如何如何。忽然间就出现了点对点会议、点对点创投基金。甚至还出现对点对点的激烈反击，看到一群白痴商业记者快乐地互相抄袭报导：「点对点已死！」

架构航天员会说会说出类似这样的话：「你能想象一个像Napste「这样的程序，你可以用它下载任何东西而不只是歌曲吗？」然后他们会建立像Groove这种自认比Napste「更一般性的应用程序，不过似乎却忘了那个微不足道，让你输入歌名然后听歌的小功能，那个我们从一开始就要的功能。说到搞不清楚重点。如果Napster不是做成点对点形式，却的确能让你输 入歌名就能听歌，恐怕还是会一样流行。

另一个架构航天员都喜欢做的事，就是发明某种新架构然后声称它可以解决某件事。Java、XML、Soap、XmlRpc、Hailstorm、.NET、Jini…天啊，我都要睡着了。而这些还只是过去12个月的而已！

我绝对不是说这些架构有什么问题。它们都是相当好的架构。让我受不了的是围绕在架构周围那些惊人的超级宣传。还记得微软.NET的白皮书吗？

新一代的Windows桌面平台Windows.NET支持生产力、创造力、管理、娱乐以及更多更多，是 为了让用户掌控其数字生活而设计的。

这东西大概是九个月前的事。上个月我拿到微软的Hailstorm。那份白皮书写着：

人们并没有掌控周边所围绕的科技…HailStorm让生活中的科技相互结合，在你的掌控之下 为你服务。

噢，了不起，所以现在你房间里的高科技卤素灯不会再随便乱闪了。

微软并不孤单。下面这一段是摘录自Sun Jini白皮书：

这三个事实(你是个新世代的系统管理员，嵌入式微电脑隐身于周遭，单机计算机无所不在)应该结合起来，以改善使用单机计算机的世界-借着消除各种计算机间的界限，借着让计 算机无所不在，借着把用计算机工作的细节变得像把DVD放进家庭剧院系统一样简单。（译注: 请原谅我，因为这一段我原文也看不懂）

还有那段让我根本不想想起来，由业界推手George Gilde「(译注：美国科技趋势专家)宣扬Java的话：

科技史上的一个十分重要的突破…

这是一个明确的情报，显示架构航天员正在攻击你：数不尽的夸大言辞；史诗般理想化的豪言壮语；夸大；完全缺乏真实感。可是大家就是吃这套！这些商业报导真是疯了。

大家究竟为什么会受无聊的架构所影响呢？这些架构常常只不过是一种用于RPC的新格式或是新的虚拟机罢了。这些东西可能是不错的架构，也的确能帮助开发者，不过它们并不(我 得强调不)能代替弥赛亚骑白驴进耶路撒冷或是世界和平。不，微软，计算机不会突然开始 会读心术然后自动做我们想做的事，只因为世界上每个人都有一个Passport账号。不，升阳， 我们并不想让公司业务资料的分析像「把DVD放进家庭剧院系统一样简单」。

要知道搞架构的人会去解决那些他们能解的问题，而不是那些解了会有用处的问题。Soap + WSDL或许是很热门的新玩意，不过它并不会真的让你做到那些以前用其他技术做不到的事(如果你真要做的话)。架构航天员鬼扯的这些分布式服务天堂过去都曾有人答应过，如果 用过DCOM或JavaBeans或OSF DCE或CORBA的话就知道了。

我们现在可以用XML作为电话上用的格式，这的确是很不错。值得欢呼一声。不过那对我来说，就像知道超市用卡车由仓库运货来差不多有趣。打个哈欠，是芒果哦，很有趣。讲些我以前做不到而现在可以的新鲜事吧，老航天员。否则就乖乖呆在太空，不要再浪费我的时间了。

十五、开火与运动

我总会有时候什么事都做不了.

我当然还是会去上班，不过却是到处闲逛，每10秒就收一次信，逛逛网站，甚至做些付信 用卡账单之类不用动脑的事.什么都做就是没法子进入状况回来写程序.

这种不事生产的毛病通常一发作就是一两天.不过在我职场生涯中也曾有过几个星期没有 产出的情况.就像别人所说的一样，我没有进入状况.我定不下心来.我不知道自己在干啥.

每个人的情绪都会波动；有些人波动较少，而其他人会比较明显甚至完全失常.不过无生 产力的时期似乎和忧郁情绪有关.

这让我想到，有些研究者声称人基本上是无法控制自己吃什么，因此节食绝对不可能持久, 最后一定会回到自然的体重.或许对身为软件开发人员的我来说，同样无法控制自己的生 产力，只能接受自己有时快有时慢，并且祈祷平均起来的生产力还能让自己不会失业.

真让我受不了的是，从我做开发工作以来，平均每天只有两三个小时能有效率地写出程序. 当我暑假在微软实习时，另一个实习生告诉我他每天只有12点到5点在做事.做五个小时 (还要扣掉午餐时间）的事，而他的小组却很崇拜他，因为他的成果还是比一般人多很多. 我也发现一样的情形.当我发现大家都非常努力工作，而每天只有两三个小时效率的我却 还是组里产出最多的人，不禁令人有点内疚.这或许就是Peopleware和极限程序作业(XP, eXtreme Programming)紧持不加班而且每周只应工作40小时的原因吧，他们很确定这样并 不会降低团队的效率.

不过一天”只有”两三小时有工作并不是问题，真正困扰我的是那些什么都没做的日子.

这件事我想了又想.我尝试回想我职场生涯中最有效率的时候.极可能是在微软把我放到 一间漂亮豪华的新办公室，里头有着如画般的大片窗景(美丽的石头庭园，里面满是盛开的 樱桃树）.那时候每件事都很顺利.在几个月内我马不停蹄地写出Excel Basic的详细规格 –非常厚的一份文件，巨细糜遗地叙述一个庞大的对象模型和程序开发环境.过程中完全没 有停顿.有一次不得不去波士顿参加MacWorld,我还是带着手提电脑坐在哈佛商学院的漂 亮阳台上，继续写着窗口类别的文件.

当你进入状况后，要继续维持并不算太难.我的一天通常都是这样子的：（1)上班（2)看 信看网页等等（3)决定应该吃过午饭后再做事（4)吃完午饭回来（5)看信看网页等等 (6)终于决心该开始干活（7)看信看网页等等（8)再度下定决心真的该开始做事（9)把 该死的编辑器叫出来然后（10)不断地写程序直到突然发现已经下午7点半了 .

第8步和第9步之间似乎有点问题，因为我不是每次都能顺利跨越鸿沟.bike trip对我来说, 要开始本身就是唯一的难题.静者恒静.我脑袋里有些东西重得不得了，很难很难加速， 不过一旦全速运转就不必费心维持.就像要骑车来一趟横跨美国的自助脚踏车旅行一样– 当你骑上车要开始时，根本无法想象要花多少力气，不过一旦开始骑就发现实在是轻而易举.

或许这就是生产力的重点：开始做吧.或许配对程序作业所以有效，是因为把两个人排在 一起作业，自然会强迫彼此开始.

当我还在以色列当伞兵时，有位路过的将军给我们讲了一小段关于策略的课.他告诉我们， 在步兵作战时只有一种策略：边开火边移动.你要一边开火一边朝敌人挺进.开火让敌人 抬不起头，不能向你开火.（这就是士兵喊"掩护我"的意思.也就是说"对敌人开火逼他低 头，这样我冲过街时敌人就不能向我开火"这招的确有用.）移动让你攻占地盘并且更逼 近敌人，这时候射击更容易打中目标.如果你不移动，敌人就能控制局势，这可不怎么好. 另外如果你不开火，敌人就会对你开火把你钉在原地.

这件事我一直记得.我也注意到由空中缠斗到大规模的海军演习，几乎所有军事策略都是 由边开火边移动的概念延生的.我又花了 15年才了解这也是在生活中成功的方法.每天你 都得前进一点点.你的程序不好有错还是没人要，这全都没关系.只要你一直进行，持续 的写程序并修正错误，时间就会站在你这边.当竞争者对你开火的时候要注意.他们可能 只是想逼你忙着应付，让你不能继续前进.

想想看微软所推出资料存取策略的历史吧.ODBC, RDO, DAO, ADO, OLEDB,还有最新的 ADO.NET -全部都是新出的！难道这些技术都是非要不可的吗？还是一个年年都在重新发 明数据存取的无能设计团队的杰作呢？（这很可能是真正的答案.）不过最终的结果却刚好 成为火力掩护.它让竞争者别无选择，只能用尽所有时间进行移植和升级，没有时间去写 新功能.仔细看看软件业界.成功的公司对大公司的依赖最少，不需要花所有工夫追随并 重新实作，然后去修那些只出现在Windows XP上的问题.而跌跌撞撞的公司都花太多时间 去揣测微软未来的方向.大家都担心.NET的出现，认为有绝对必要所以决定针对.NET重写 整个架构.事实上微软是在对你开火，而且只是让他们前进并阻碍你们的掩护火力，因为 这就是游戏规则，朋友.你想支援Hailstorm吗？ SOAP呢？ RDF怎么样？你支持这些东西是 因为客户需要？还是因为有人对你开火而觉得应该有所反应呢？大公司的业务团队很了解 火力掩护这一套.他们会去跟客户说"没错，你不一定要买我们的东西.要买就要买最好的. 不过记得你买的产品一定要支持(XML /SOAP / CDE / J2EE),否则你就会被绑住了然后 当小公司试图接触这个客户时，这个听话的技术总监就会像鹦鹉一样说"你们支持J2EE吗？" 尽管J2EE不会真正带来收入，他们还是得耗尽所有的时间加上J2EE,结果完全没机会让产 品产生区别.这是个勾选项目–会去做只是因为需要有个项目打勾表示你也有，不过没 有人会用也没有人需要.而这就是掩护火力.

对我们这种小公司来说，边开火边移动有两个意义.你必须争取时间，另外每天都得 要前进.你迟早会赢的.我昨天整天只是把FogBUGZ的配色改善了一点点.这并不要紧.东 西会愈来愈好.我们的软件每天每天都会变得更好，而且客户会愈来愈多，这就够了.在 我们变成Oracle这种规模的公司前都不用管什么伟大的策略.我们只要每天早上来公司， 想办法要自己打开编辑器就好了.

十六、人员技能

我想想来谈谈我为CityDesk 3.0重写的一段程序：档案汇入程序。（广告：CityDesk是本公司出品的很容易用的内容管理产品。）

它的规格似乎和任何一段程序代码一样简单。使用者用标准对话盒选择一个档案，然后程序将该档案复制到CityDesk的数据库中。 结果却变成「最后1%程序用掉90%时间」的最佳范例。最早一版的程序结构大概是这样子的：

1. 开档案
2. 把内容全部读进来，放在一个大字节数组里
3. 把字节数组放在一个记录中

效果不错。大小合理的档案几乎是一瞬间就做好了。它有几个小问题，我们一个一个来看。

当我把一个120MB的档案丢进CityDesk时跑出第一个问题。通常人们不太会在网站上放个120 MB的档案。事实上应该是少之又少才对。不过虽然少但还是有可能。程序可以用，不过等了一分钟而且画面上完全没有反应-程序就这样定住不动，看起来好像完全当掉了。这显然 并不太理想。

由使用接口的观点来看，我真正要的是在长时间处理时出现一个进度棒之类的东西再加上一个取消钮。理想的状况下，你可以在CityDesk里继续其他操作，档案复制的作业会在背景执行。这有三种显而易见的方法可以达成：

1. 只用一个线程，由该线程经常轮询输入事件
2. 启用第二个线程，然后小心的进行同步
3. 启用第二个行程，然后不必那么小心的进行同步

我的经验是第一种作法行不通。要确定应用程序里所有程序代码都能在复制档案时安全的执行，实在是太困难了。另外Eric S. Raymond也让我相信，以解决方案来说线程通常不如单 独的行程好：事实上多年的经验显示，多线程程序设计产生了很多额外的复杂性，而且会引 入全新种类的危险海森堡虫(heisenbugs)。第三项似乎是个好答案，尤其当我们下层用的是 多用户的数据库，不用怕很多行程同时使用。所以我打算等感恩节休假结束就用第三种作法。

不过整体想想看就会发现，我们已经由读文件写作数据库变成复杂许多的东西：启动一个子行程，让它读文件写入数据库，这个子行程还要有一个进度棒和取消钮，最后还要某种机制让子行程通知母行程，档案已经处理完可以显示..其他的工作还包括将命令行参数传给子行程，确定窗口焦点的行为正确，还要能处理使用者在复制档案时关机的状况。我猜等把这些都做好，大概要十倍的程序才能优雅地处理大档案，而这可能只是用户所看到的程序的百分之一。

当然会有某类型的程序员出来争论，说我用子行程的新架构比原本的差。这种作法「过度膨胀」（由于那些额外增加的程序代码）。另外潜在的问题也比较多(一样由于是多加的程序代码）。这个作法做太过头了。这某种程度也表示Windows的确是个烂操作系统。他们会说这和 进度棒有什么关系？他们会嘲笑着说：只要按Ct「I+Z再重复打”ls -l”看看档案大小有没有增加就好啦！

这个故事的教训是修正一个1%的问题可能会用掉500%的工夫。这种状况并不只软件业会发生。 敢这样讲是因为我现在就在管理这些建筑工程。上星期我们的包商终于完成Fog Creek新办 公室的最后修饰工程。内容包括在前门安装闪亮的蓝色压克力，旁边围着铝框，铝框上每20 公分就镶有一颗螺丝。如果你仔细看照片，会发现每片门都会被铝框整个包住，当门关起来 时，两片直框会并排在一起。虽然图上看不出来，不过中间铝条上的螺丝几乎但没有确实对 齐。差了大概有2公厘。负责这件事的木匠很小心的测量，不过装铝框时门是放在地上，并 不是等好门装好再当场装的。所以等到要把门装上去时，才发现螺丝并没有对齐。

这或许并不罕见；；我们办公室里很多门的螺丝都没有对的很好。问题是当洞钻好之后，要重 新修正的代价贵得离谱。由于正确的螺丝位置离原来的洞只有几公厘，所以不能直接在门上 再钻一个，可能得把整个门换掉才能解决。这实在是很不值得。这正是另一个修正1%的问题 会用掉500%工夫的例子，而且这也解释了为何这世上很多人造物都是99%好而非100%完美。 (不过建筑师倒是不断吹墟，说某些亚历桑那州的高贵金屋里每根螺丝都排得很整齐。）

这点出了软件的一项特性，一项大部份人都视为工匠技艺的特性。当软件是由真正的工匠制作时，所有螺丝都会对得整整齐齐..当你做些很罕见的事时，应用程序会表现得很聪明。作 者会投入更多的工夫去正确处理特的状况，而不光是让主要的程序会动。即使要额外再花500%的力气去处理1%的状况也在所不惜。

工匠技艺当然是非常昂贵的。唯一负担得起的方法就是针对大量的客户开发软件。很抱 歉，不过保险公司开发的内部人事管理程序绝对不可能达到这种工艺的境界，因为就是没有 足够的使用者来分散额外的成本。不过就软件包公司而言，这种工艺境界可以让使用者高兴 并提供长期的竞争优势，所以我愿意花时间并正确地执行。请容忍我吧。

十七、源于计算机学科的三个错误思想

我并不是要泼大家冷水，不过坦白讲信息科学里有三个重要的想法是错的，而且大家也开始注意到了 ..为了你自己好请忽略它们。

我相信还有其他的，不过这是最让我感冒的三个：

1. 搜寻的困难之处在于找到足够的结果，
2. 去银齿(anti-aliased)的文字比较好看，还有
3. 网络软件应该让网络资源的行为和本地资源一模一样。

嗯，我只能说，

1. 错，
2. 错，
3. 全部都错！

让我们来个简短的巡礼。

搜寻

大部份搜寻的学术研究都一定会被类似的问题所困扰：比如「如果搜寻”ca「”，可是想要的文件内用的却是”automobile”，这时要怎么处理？」

事实上极大量的学术研究在探讨多形态(stemming)之类的概念，在这个概念下你所搜寻的字会被去结合（de-conjugated)，所以用”searching”来搜寻时，也会找到内含”searched”或“sought”的文件。

所以当Altavista这种大型Internet搜索引擎刚出现时，都在宣传他们可以找到很多很多的 结果。用Altavista搜寻Joel on Software会得到1,033,555个网页。这种结果当然是无用的，

目前已知Internet大约有十亿个网页(译注：当时是公元2000年），Altavista把十亿个网页 缩减成一百万个，对我来说是完全没有意义的。

搜寻真正的问题在于如何将结果排序。不过要为计算机科学家辩护一下，除非有人开始拿 I nternet这么巨大的数据来做索引，否则根本不会有人注意到这一点。

不过还是有人注意到了。Google的Larry Page和Sergey Brin了解到，以正确的顺序呈现网 页ranking，比把所有可能的网页都抓来要重要多了。他们的PageRank算法是个很好的方法，可以对很庞大的结果进行排序，让你所要的结果很可能列在前十项。事实上用Google搜寻Joel on Software就会看到它出现在第一项。用Altavista搜连前五个网页都不是，我都懒 得去找究竟排在哪了。

去锯齿的文字

去锒齿处理是1972在麻薯工学院的Arch itecture Mach i ne Group (后来合并到著名的Med i a Lab)发明的。它的想法是针对低解晰度的彩色显示，可以用灰阶来产生分辨率的「幻觉」。下面是它看起来的样子：

注意左边的正常文字漂亮又锐利，而右边去锯齿文字的边则显得模糊。如果你斜着看或是退后一点看，正常文字会因为计算机解晰度有限而出现怪异的「阶梯状」。而去锯齿的文字看起来反而比较平滑好看。 所以这就是大家对去锯齿都很兴奋的原因。现在到处都会用去锯齿，甚至连微软Windows都 有一个开关可以打开所有系统文字的去锯齿效果。

那问题呢？如果你尝试去读一段去锯齿的文字，感觉起来就是模糊。事情就是这样，我也没法子。比较一下这两个段落:

左边的段落没有去锯齿；右边的则是用Corel PHOTO-PAINT去锯齿的段落。坦白地说，去锯齿的文字看起来就是不好。

终于有人注意到这个问题了 ： M icrosoft Typography group。他们创造了一些非常好的字型， 像是Georgia还有Verdana等等，都是针对容易在屏幕上阅读而设计的。他们基本上放弃先创 造高解晰度字型再塞入像素格子的作法，而是把像素形成的方格视为先天限制，然后依照限 制设计一个能配合的字型。不过也有些人没有注意到这件事：Microsoft Reader group使用 了另一种他们称之为”ClearType”，针对彩色LCD屏幕设计的去锯齿技术。不过很遗憾的看起 来还是模糊，即使在彩色LCD屏幕上看也一样。

(在读者中有绘图专家生气抗议之前，我应该要先澄清一下，就标题和图案而言去锯齿仍然 是个很好的技术，因为这些应用的整体外观比维持可阅读性更重要；另外对照片也很有用， 去锯齿可以有效地把照片缩成较小的尺寸。)

网络透明化 远从第一个网络开始，提供让存取远程资源和本地资源完全相同的程序接口就一直是网络运 算的「圣杯」。有了它就可以让网络变成「透明的」。

网络透明化有个著名的例子RPC (远程过程调用），这个系统能让你呼叫在网络上另一台计算机执行的程序(子程序)，就像这些程序是在本机计算机执行一样。大家在这上面投入了相当多的工夫。另一个例子是建立在RPC之上的微软分布式COM(DCOM)，它可以存取在另一台计算 机上执行的对象，就像对象是在自己的计算机上执行一样。

听起来很合逻辑，对吧？

错。

别台机器资源和本地机器资源在取用上有三个非常大的差异：

1. 可使用性(Availability)，

2. 延迟（Latency),以及
3. 可靠性(Reliability)。

当你要取用别台机器的资源时，很有可能该机器或是网络无法使用。其次网络的速度意味着需要一段时间去处理该要求，比如你可能是透过28.8kbps的调制解调器执行的。另外对方的机器可能会当掉，网络也可能会在通讯过程中断线（比如猫突然跌倒压到电话线）。

可靠的软件在使用网络时绝对都得考虑这些问题。程序设计界面把这些都隐藏起来，对写烂软件来说真是帮助匪浅。 来个简单的例子：假设我的软件要把档案由一台计算机复制到另一台。Windows平台中传统 的「透明」作法，是呼叫常用的CopyFile方法，并传入\SERVER\SHARE\Filename之类的UNC 路径作为文件名。

如果网络一切正常，这种写法就会正常运作。不过如果档案大小有1 MB又是透过调制解调器连接网络，什么问题都可能发生。在传送这个1 MB的档案时，整个应用程序都会停住没有反应。也不可能显示进度，因为CopyFile当初设计时就假设一定会是「快」的。如果电话断线也不可能续传。

如果真的想透过网络传档案，用像FtpOpenFile系列函数之类API会比较好。这和在本机复制档案是不一样的，使用上也比较困难。不过这个函数在设计时就考虑到网络与本机端的程序设计是不同的，另外它提供挂入(hook)功能可以显示进度或在网络不存在或断线时优雅地处理，还可以用异步的方式运作。

结论：下次当某人想卖你一套程序设计产品，声称可以让你存取网络资源有如本机资源 一般，请往反方向全速逃离。

十八、二元文化

到目前为止，Windows和Unix在功能上的相似之处比不同的要多。他们都支持同样的主 流程序作法（由命令行到图型接口到web服务器）；他们所依循的系统资源基本是完全一样的，比如几乎完全相同的文件系统、内存、sockets，还有行程和线程等等。两套操作系统的核 心服务，还有开发者能写的应用程序类型也没多大的差别。

剩下的就是文化差异。没错，我们全都要吃东西，不过海那边的人是用木筷子吃生鱼片 加米饭，而我们在这里则是用手拿着面包夹大块碎牛肉在吃。文化差异并不表示美国人的胃 不能消化寿司，或是说日本人不能消化大麦克汉堡；也不表示吃寿司的美国人或吃汉堡的日 本人很少；文化差异指的是美国人第一次去东京下飞机时所面对，「这地方真奇怪又该死」 那种无法承受的感觉。至于什么「我们其实都是一样的，我们都有爱，也都会工作欢唱及死 亡，将会克服美国人和日本人永远无法真正对对方的厕所安排感到自在。」这种空话根本连 想都不会想。

Unix和Windows程序员的文化差异是什么呢？细微的地方很多，不过有一项最明显:Un ix 文化重视对其他程序员有用的程序，而Windows文化重视对非程序员有用的程序。

这当然是主要的简化说法，不过事实上确还是个很重大的差异：我们写程序究竟是为程 序员还是为了 一般使用者？除了这个问题之外其他的都只是诠释。

时常大发议论的Eric S. Raymond刚写了一本有关Unix程序设计的巨作The Art of UNIX Programming，极其深入地探索他自己的文化。你可以去买这本书的纸本来看。如果你觉得Raymond的想法太过激进(anti-idiotarian)而不想付钱给他，还可以在在线免费阅读，保证 作者的心血绝对不会收到一分钱。

让我们来看个旋子。Unix的程序文化格外重视可由命令行呼叫的程序，这种程序可以 用参数控制各种动作，其输出可以撷取成一般机器可读的纯文本格弋。这种程序之所以会被 重视式，是因为可以让其他程序员轻易地整合进别的程序或较大的软件系统中。举个很小的 例子，Unix文化中有一个Raymond称之为「沉默是金」的核心价值，是说当程序成功地完成 你所交付的事时应该不要有任何输出。不管是在命令行输入300个字符的命令建立文件系统，还是建立并安装了一套复杂的软件，或是把载人火箭送到月球，全部都一样。只要成功执行就不要有任何输出。使用者看到下一个命令行提示就会推论一切正常。

由于你是为其他程序员写程序，所以这才会是Unix文化的重要价值。就如同Raymond所 说的：「喋喋不休的程序不太能和其他程序好好合作。」相较之下在Windows文化中是为张 阿姨而写程序的，而张阿姨搞不清楚程序是因为执行成功而没有任何输出，还是因为出错了 所以无法显示任何讯息，或是因为误解你的要求而没有输出。

同样的状况，Unix文化欣赏保持文字接口的程序。他们不是很喜欢图型接口，不过在纯 文本接口程序上漂亮地涂一层GUI 口红可以例外，另外他们也不喜欢二位文件格式。因为文 字接口比较好用程控，而图型接口除非提供其他机制(如内建脚本语言），否则几乎不可能用 程

控。我们在这里又再次看到，Unix文化重视有助于其他程序员的程序，而这一点很少会是Windows程序设计的目标。

这并不是说所有的Unix程序都只是为了程序员而设计的。这样说太偏颇了。不过这个文 化重视对程序员有用的东西，而这可以解释很多很多的事情。

假设你找了一个Unix程序员和一个Windows程序员来，要他们分别写同一只针对一般用 户的应用程序。Unix程序员会先建立一个由命令行或文字接口驱动的核心，通常事后才写一 个图型接口来呼叫核心。这样子应用程序的主要功能都可以让其他程序员使用，他们可以用 命令行呼叫程序并把结果以文字方式读入。而Windows程序员会倾向先由图型接口开始，通 常事后才加上脚本语言让图型接口的操作自动化。对一个99.999%用户完全不写程序(也不打 算学写）的文化来说，这是很正常的。

在Windows程序员中有个主要针对其他程序员写程序的重要团队，就是微软内部的 Windows团队本身。他们做事的方法通常是建立一个可以由C语言呼叫的API，在API里实作出 功能，然后再撰写图型接口程序来呼叫API。所有用Windows使用接口能做的事，全都可以用 任何合理的程序语言呼叫某个程序接口做到。举例来说，微软的Internet Explorer本身只 不过是一个89 KB的小程序，里面包装了数十个可供程序老手免费使用，功能强大又有弹性 的的组件。问题是程序员无法接触到这些组件的原始码，所以只能完全依照微软内组件开发 者所预见并允许的方式使用，而这种作法并不是每次都行得通。有些问题因为缺乏API原始 码，以致于难以甚至无法追查，这时通常就怪罪到呼叫API的人身上。Unix具有公开原始码 的文化价值，使它成为更容易开发的环境。每个Windows平台的开发者都会告诉你，某一次 花了四天去查一个问题，原因只是他们认为LocalSize传回的内存大小应该和原先呼叫 LocalAlloc时用的值一样，或是其他有链接库原始码就能在十分钟内搞定的问题。Raymond 创作了一个有趣的故事，说明这个曾经用过无原始码链接库都会同意的道理。

所以你看到这些宗教性的论点了。因为可以进入链接库除错所以Unix比较好。因为张阿 姨可以看到提示确认她的电子邮件真的送出去了，所以Windows比较好。事实上没有谁比谁 更好，他们只是拥有的价值不同：Unix的核心价值是制作有助于其他程序员的东西，而Windows则把制作有助于张阿姨的事视作核心价值。

让我们来看看另一个文化差异。Raymond说「典型的Unix文件会写得简洁而完整]…这 种风格假设读者够积极，可以推导未写出的推论，并且有自信信任这些推导。]小心的阅读 每一个字，因为一件事很少会讲两遍。]」天啊，我认为他其实是在教年轻的程序员写出更 不人性的man说明文件。

这对一般使用者而言是绝对行不通的。虽然Raymond可能说这是「过于简化的屈就]」， 不过Windows文化了解一般使用者不读文件，即使不得不看也是愈少愈好，因此你必须不断 重复的解释…事实上在良好的Windows说明档中，必须每个主题都能让一般读者直接看懂， 不必先看懂其他任何的帮助主题。

怎么会出现不同的核心价值呢？这是Raymond的书另一个出色之处：他深入Unix的历史 和演进，让新程序员熟悉回溯至1969年起累积的历史。在Unix被创造出来及其文化价值成形 的时候，一般使用者并不存在。计算机很贵CPU时间也很贵，学计算机就等于是学写程序。 无怪

会蕴酿出重视有助于其他程序员的文化。相较之下Windows的创造只有一个目标：尽量 卖更多套以获得更多利润，成亿上兆套的卖。「每个家庭每个桌上都有一台计算机」就是创 造Windows的明确目标，并且设定了它的议题也决定它的核心价值。让非程序员容易使用， 是进驻每个桌面和家庭的唯一方法。因此使用性自然凌驾其他东西而成为文化的基准。对这 样的文化而言，程序员是极其次要的对象。

文化的裂缝是如此的明确，所以Unix从未真正的进入桌面。张阿姨不会真的去用Unix。 不断有人努力帮Unix制作漂亮的外壳，希望让张阿姨能用Unix，不过全都失败了。因此这些 程序员都深陷在Unix文化中。举例来说，Unix有一个源自于X设计者的价值：策略与机制分 离。这种想法直接导致使用接口的分裂；桌面使用接口运作的细节并没有大家一致同意的做 法，由于这些设计者的文化重视这种差异，所以他们就认为这种作法可以用。不过对张阿姨 来说，在不同程序里剪贴得用不同的使用接口，绝对是行不通的。所以这就是现在的状况，Unix开发者20年前开始尝试在系统表面涂上一层漂亮的使用接口，结果到今天最大Linux厂 商的执行长还告诉大家，家庭用户应该直接用Windows就可以了。我曾经听过有经济学家宣称，说硅谷永远不可能在其他地方（比如说法国好了）重现，因为法国文化对于失败的惩罚太 重，以致企业家不愿意冒险。或许Linux也有相同的状况：Linux可能永远都不会成为桌面的操作系统，因为它重视一些会阻碍这件事的价值。OSX就是一个证明：苹果终于创造了给张阿姨用的Unix，不过原因并不光是因为苹果的工程师和经理坚持针对一般使用者的文化 (我独断地称之为「Windows文化」，虽然这从历史来看是源于苹果），他们还拒绝了Unix文 化中以程序员为中心的基本思想。甚至把核心目录的名字改掉了（多么违反传统！），用常见的英文单字”applications”和”l ibrary”代替”bin”和”lib”。

Raymond真的尝试比较Unix与其他操作系统并突显之间的差异，不过这却也是这本好书 中最弱的部份，因为他真的不知道自己在讲什么。当他谈论Windows时常会显露他对Windows 程序设计的知识多是由报纸读来，并非来自真正的Windows程序设计经验。这没什么关系： 他并不是Windows程序员，而这一点我们可以体谅。不过他身为对一个文化有深厚了解的典型，知道该文化所重视的价值，却没有注意该文化中针对一般人部份(杀老太太跟当机：一 定是不好的），与针对程序员部份(吃生鱼片和命令行参数：因对象而不同）的差异。

有太多的单文化程序员，他们就像从未离开过家乡的典型美国小孩，说不出来文化价值 和人类核心价值间的差异。我遇过太多嘲笑Windows程序设计的Unix程序员，他们认为 Windows野蛮又愚蠢。Raymond经常落入未考虑原由就轻视其他文化的陷阱。在Windows程序 员中很少会发现这种顽固份子，因为Windows程序员大体上都是以解决方案为准而非意识型 态的。至少Windows程序员会承认他们文化的缺陷，然后很实务地说：「看，如果你想把字 处理程序卖给很多人，就得能在他们的机器上执行。如果得用邪恶的registry代替优雅的 ~/.「c档储存设定才能办到，就这样做吧。」现在Unix世界充满自以为是的文化优越拥护者 和s l ashdot的灌水教派（karma-whor i ng sectar i an i sm)，而Wi ndows世界比较实务（我毕竟得 在这里讨生活嘛）。这种现状源于Un i x文化感觉自己被围攻，无法突破服务器和爱好者小众 市场进入主流桌面市场。这种弱势的傲慢正是The Art of UNIX Programming最大的缺点， 不过倒也并不是如此严重，整体而言这本书充满了非常有趣的洞见，内容泛及各种程序设计 主题，以致我愿意闭住呼吸忍受偶而出现的意识型态高论，因为在其他部份可以学到太多一 般性的观念了。事实上我会把这查书推荐给任何平台任何文化和任何目标的开发者，因为里 面有大

多价值是普世通用的。Raymond指出CSV格式不如/etc/passwd格式时，他尝试由Unix 而非Windows观点去评价，不过你知道吗？他是对的。/etc/passwd比CSV容易分析，如果你 读了这本书就会知道为什么，而且也会成为更好的程序员。

十九、自动获取用户故障报表

我曾经受雇于在美国一间最大的互联网服务供货商撰写客户端应用程序，所开发的产品拥有数以百万的用户,即使一个最罕见的错误都可能影响数百甚至数千的用户。虽然如此,每次当 我们决定按下按钮以释出最新版本的程序时我还是充满信心。我还记得我曾经告诉我爸「这 次的试用版本看起来棒极了。昨天我们在北美洲只收到十二个错误报告。」

十二个，嗯?还是十三个？

不，是十二个。我们应用了一种日渐普及、由客户端自动报告错误的技术。所得的报告都记录到错误追踪数据库内汇总成摘要,以便开发人员寻找并修正只在客户端发生的错误。这类 错误往往难以在我们的测试实验室内测得到，因为事实上我们无法全部设置或模拟成客户所 有超乎寻常的计算机组合。

错误侦测及报告功能正日渐普及。现在互联网浏览器和窗口系统都已经内建这类功能，而且越来越多客户希望他们的桌面应用程序懂得自行回报所遇到的错误。

能够知道世界上每一个角落找到的每个错误，对于建立一份你所开发的产品能在这个疯狂的世界上安然无恙的信心而言非常重要。尤其对于像开发消费者应用程序领域的我们而言更 为重要。很多时候你不能倚靠客户告诉你他们所遇到的错误，他们可能没有足够的技术背景, 更重要的是，要是没有自动报告错误的机制，他们根本不会有兴趣在忙着干自己重要的工作 时还抽空给你发一份完整而有用的错误报告。

现在我创立了自己的公司,更深深地相信自动收集客户端错误的重要。差不多所有我们在Fog Creek Software编写的程序都有好些方法能透过我们的FogBugz错误追踪数据库回报错误。我们的产品CityDesk之内的产品，以至FogBugz本身(一个安装在客户端的网页服务器上的 网页应用程序)都内建了这项功能。两者皆可以透过互联网汇报错误。就连我们为内部开发 的应用程序，比如说营运Fog Creek网上商店的电子商务程序，都会用相同的方式通知开发团 队程序运作期间所遇到的错误。

剖释错误

好吧，你的程序当掉了。在几乎每个程序执行环境下,总有些方法能让你由某个地方回复、重新开始作业。以往我们只会让程序在出问题的的地方乖乖挂掉,现在我们选择弹出一个对话框。 这么多年来我学到一件事，就是你所问的问题越多，想回答的人就越少。所以现在我们只会问 最少的而又能帮助分析错误的问题。比如说「你正在做什么？」、「你的电邮地址?」我们 强调提供电邮地址是自愿性质,以排解关于个人隐私的疑虑。令人意外的是有些盲目恐惧是 如此深入民心而且根深蒂固。在消费者市场内，许多人都已经被十一点钟新闻报导教育成永 不要交出他们的电邮地址以避免收到垃圾邮件。结果是,如果你需要人们汇报错误的时候填写电邮地址，他们就干脆不提交报告。

基本上所有你所需要的数据都可自动取得,例如操作系统的版本,系统拥有的内存大小等等。

数据收集

接下来的问题是，要收集什么样的数据才可以帮助我们的开发人员找到造成错误的地方?我 们很难抗拒把所有抓得到的数据都抓起来的这种诱惑。抓下每个你可以抓到的位信息，抓下 用户系统内所有DLL及COM控件的版本,以至把所有内存内的数据都倒出来制作一个备份(有 些会叫这做核心转储Core Dump)。

然而在做了开发人员几年后、以及在从不知道自己拿着份核心转储数据可以干些什么的情况下,我终于发现这些都是非必要的数据。以下都是我们实际上会收集的数据：

- 产品版本
- 操作系统版本及微软Internet Explore「版本(有相当多的窗口功能都是来自Internet Explore「和它的组件，以致这项数据对于了解包括图象接口应用程序在内 的程序对窗口操作系统上的表现非常重要)
- 发生错误的程序文件以及程序代码行数错误消息正文
- 独一无二的错误码
- 来自用户关于他们正在进行中的作业的描述
- 用户的电邮地址

有这些就足够了！这些年来我们发现单是知道发生错误的程序代码行数就己经足够帮助修正大部份的问题。在少数以这项信息不足以解决问题的个案上，你可以透过直接联络遇上问题的用户以取得更多的信息帮忙解决问题。收集少量信息的好处在于令回报错误的程序变得非常快速，进而减少用户对回报问题感到不耐烦的机会。单是检查所有DLL和COM控件的版本就 可以花上好一段时间,如果数据是由调制解调器上传的话就又得花更多的时间,而这些数据是可以提供帮助修正错误的有用资讯。即使你发现错误只会在程序应用一个微软系统DLL的某个版本时才会发生,你又可以做些什么?你仍然需要改写程序来避开错误。

致电回家

感谢上天互联网的无处不在，在大部份情况下透过网页传送信息都是把信息送回家的最佳方法。只要送出一个标准的HTTP需求，你的错误报告几乎可以穿越于客户环境内任何种类的防火墙。更妙的是，现在几乎每一种编程环境都有内建的链接库支持发送HTTP需求及读取响应。 举例来说，在窗口平台上你就可以在WININET链接库内找到内建的函数式去利用Internet Explore「的网络传输程序代码来传送HTTP需求和读取响应。利用这些函数式最好的是，如果 用户曾经设定他的浏览器透过代理服务器来在防火墙内传取数据，WININET呼叫程序会自动 采用相同的设定操作，而过程中你无付出任何额外的努力更改或设定你的程序就能正常发挥 功用。

我们的FogBugz在收到HTTP需求送来的错误报告后，会传回一个很短的XML文件表示已经收到 错误报告，文件亦同时包含一个展示给用户的讯息。关于这点，稍后我将会谈得更多。

如果你的应用程序是一个网页浏览器应用程序，你可以采用一个更简单的做法：展示一个包含一张可递交数据到服务器上的表格的网页。你只需要将表格的行动路径指向你的错误追踪数据库即可。请参考图象2。

在好些种类的应用程序当中，你或许会想将错误报告先写到档案或登录内，等到用户重新启动应用程序时才送出，而非在错误发生时直接送出。我称呼这种技巧为延迟传送。这种做 法虽然会延后收到报告的时间，但好处是一旦发生的错误严重至令你的应用程序瘫痪到无法 送出错误报告时，你仍然可在稍后取得报告。

现在所有要送交Fog Creek的错误报告都会送抵一个位于我们公众服务服务器上的URL。而我们的错误追踪数据库则由透过这个URL接收错误报告。事实上这个URL是公众可以接触到错误追踪数据库的唯一路径。除了接收报告以外的所有功能都已被死锁，客户可以提交错误报告，但无法读取数据库内的任何数据。

图象3展现的是当一个错误报告抵达我们的数据库时的模样。我们可以设定将收到的报告自 动送给开发团队内的其中一位工作人员，不过最近为免打扰同事，我们选择将报告都送到 被建立的虚拟人”CityDesk New Bug”手上。每次我们想要过滤出收到的报告，只需找出那些 被分派到虚拟同事手中的错误，然后再逐一决定是否需要跟进修正。所有需要修正的错误将 会被派到真实存在的同事手上。

办认重复的错误

应用自动错误报告收集时很重要的一点是，同一个错误可以在许多不同的人手上发生许多次， 而你绝不会希望每次同一个错误被报告时都会在你的错误追踪数据库内制做一个新的错误 记录。我们应付这个情况的方法是以错误信息内的重要数据为错误报告编上独特的标识符串。

字符串的编码方法十分谨慎，分别来自两个用户的同一错误必须获编相同的标识符串。在好些实验以后，我们发现最好的编法就是在字符串内加入错误码、文件名、函数式名称、错 误行数和产品版本。图象3内展示了标识符串”E「「o「91 (global:IsRoot:0) V1.0.32”。字 符串说明档案global.bas于执行函数式IsRoot位于第0行的程序代码时发生91号错误，而产 品的版本为1.0 build 32。相当偶然地，我们总是将编译版本数目为双数的产品编为内部使 用版本，而编译版本数目为单数的则是会送交到客户手上的版本，这样我就可以一眼知道某 个错误报告到底发生在开发人员或客户身上。

FogBugz往后遇到标识符串相同的当机回报时，会自动附加到这个案例中，并不会重开一个新案例。这样能让程序员在同一个地方看到所有相同的当机状况。

标识符串的格式设计得花点心思。我们以前会把当机错误讯息整个含在标识符串里。不过很快就发现错误讯息会被译成不同的语言，于是每种错误都会分散成英德西法及其他几种我 认不出的语言。我们解决的方法是把错误讯息放在当机报告内，但是把不随语言而变的错误 标识符放在报告的标题中，这些重复的回报就少多了。

另外标题也经过特别安排，可以方便搜寻特定问题。由于我们在标题中使用「档名：函数名： 行数」的格式(注意其中的冒号），所以只要搜寻「：函数名：」就很容易找出某函数相关的 问题。我们基于相同的原因在版本号码前加上字母V，这样就能搜寻V1或V1.0或V1.0.32。如 果没有这个V字母，想找版本1时就会找出所有标题里刚好有1字符的错误报告了。

当错误被确认时，我们可以把某个旗标(FogBugz接口里的Scout Will)由「继续回报」改成「停止回报」，之后就会忽略标识符串相同的当机回报。我们甚至可以设定一串文字讯息(在FogBugz接口里是针对自动送出的案例中出现的ScoutMsg),可以自动传给往后遇到相同当 机状况的使用者。当我们要使用避开错误的作法(workaround)时也会用这个功能提出建议。 就像是「嘿，下次你要存档前要记得先拍拍自己的头再摸摸自己的肚子！」

重复回报的常见原因之一，就是当在当机处理程序本身。这并不一定是说当机处理的程序有问题，或许只是因为原先的当机太严重，以致再也没有程序能正常运行。

第二部分 开发人员的管理

二十、面试游击指南

录取合适的人对于Fog Creek软件公司来说是非常关键的。在我们这个领域，有三类人可以挑选。在一个极端，是哪些混进来的，甚至缺乏最基本的工作技巧.只要问这类人两三个简 单的问题，再读一下他们的简 历，就可以轻易地剔除他们。另一个极端的类型是才华横溢 的超级明星这些人仅仅为了好玩就用汇编语言为Palm Pilot (—种手掌计算机）写了一个 Lisp (—种人工智能编程语言）编译程序。在这两种极端类型中间的是一大群不能确定水平 的候选者，也许他们中的某些人能干些什么？这里的关键是明白超级明星和那一大堆属于中 间类型的人的区别，因为Fog Creek软件公司只录取超级明星。下面我要介绍一些找出超级 明星的技巧。

Fog Creek公司最重要的录取标准是：

有头脑，并且能完成工作（Smart, and Gets Things Done.)

就是这些了。符合这样标准的人就是我们公司需要的员工了。记住这条标准。每天上床前 背诵这条标准。我们公司的目标之一就是录取拥有这样的潜质的人，而不是录取懂某些技术 的人。任何人所拥有的某些具体技术都会在几年内过时，所以，录取有能力学习新技术的人，要比录取那些只在这一分钟知道SQL编程是怎么回事的人对公司更划算一点。

有头脑确实是一个很难定义的质量。但是让我们看一些在面试时能提问的一些问题，通过这些提问，我们可以找出拥有这种质量的人。完成工作非常关键。看起来有头脑但是不能完成工作的人经常拥有博士学位，在大公司工作过，但是在公司中没有人听他们的建议，因为他们是完全脱离实际的。比起准时完成交代事项，他们更宁愿对于一些学院派的东西沈思。这些人由以下特性而可以识别出来。他们总是爱指出两个根本不同的概念间的相似性。例如，他们会说“Spreadsheets是一种特殊的编程语言〃,然后花一个礼拜写一篇动人的，智 慧的白皮书。这篇白皮书论述了，作为一个编程语言，Spreadsheet关于计算语言特性的种 种功能。聪明，但是没用。

现在，我们来谈谈完成工作但是没有头脑的人。他们爱做蠢事。从来也没有考虑过将来得靠他们自己或者别的什么人来亡羊补牢。通过制造新的工作，他们成为了公司的负债而不是资产。因为他们不仅没有为公司贡献价值，还浪费了好员工的时间。这些人通常到处粘贴大堆的程序代码，而不愿意写子程序。他们是完成了工作，但是不是以最聪明的方式完成工作。

面试时最重要的法则是：

做决定（Make A Decision)

在面试结束时，对于被面试者，你不得不做一个直截了当的决定。这个决定只有两个结果：录取或者不录取.回到你的计算机前，立刻用电子邮件通知招聘负责人你的决定。电子邮件 的主题应该是录取或者不录取。接着你需要在正文中写两段来支持你的决定.

没有其他的答案。永远不要说，“？录取你，但是不能在我的团队中”。这是非常草率粗 鲁的决定，因为你在暗示应试者没有聪明到能有和你一起工作的资格，但是以他的头脑适合 进入那些天生输家队伍。如果你发觉自己被诱惑，想说出那句“录取你，但是不能在我 的队伍中〃，那么就简单的把这句话变成“不录取再说出口。这样就没事了。甚至

如果某个人在特定领域很能干，但是在别的队伍中将会表现不好，也是不录取。事物变化 的如此之快，我们需要的是在任何地方都能成功的人。如果某些情况下你发现了一个白痴专 家（拥有某些特殊能力的白痴），这个专家对于SQL非常，非常，非常的精通，但是除此之 外什么也学不会，不录取。在Fog Creek公司没有他们的将来。

永远不要说，“也许，我不确定〃。如果你不确定，意味着不录取。看，比你想象的容 易的多。不确定？就说不！同样，如果你不能作出决定，那意味着不录取。不要说，〃嗯，录取，我想是这样的。但是关于…，我想知道…”。这种情况就是不录取。

最重要的是记住这点，放弃一个可能的好人选要比招进一个坏人选还要来的好。一个不合格的求职者如果进入了公司，将要消耗公司大量的金钱和精力。其他优秀员工的还要浪费时间来修复这个人的错误。如果现在你还在犹豫，不录取。

如果你是Fog Creek公司的面试官，当你拒绝了大量的应聘者时，不要为Fog Creek公司将因此雇不到任何人了而忧虑。这不是你的问题。这是招聘负责人的问题。这是人力资源部的问题。这是Joel (译者注：Fog Creek公司的老板，本文作者）的问题。但不是你的问题。不 停地问自己，哪种情况更糟糕？ 一种情况是我们变成了一个庞大的，糟糕的软件公司，充斥 着许多脑袋空空如可可果壳(译按:火山豆，Macadamia Nuts,大洋洲特产。）的家伙，另一 种情况是我们是一个小而高质量的公司。当然，找到优秀的应征者（并聘用他们）是很重要 的。找到有头脑而且完成工作的人是公司中的每个员工的日常工作之一。但是当你作为Joel Creek公司的一员真的开始面试一个应聘者时，要装作现在正有很多优秀的人想打破头挤进 Fog Creek公司。总之，无论找到一个不错的应聘者是多么的难，永远不要降低你的标准。

但是你如何作出录取或者不录取这样艰难的决定？你只要在面试过程中不停地问自己:这个 人有头脑吗？这个人能完成工作吗？要作出正确的回答，在面试时你必须问对问题。

开个玩笑，下面我要问个有史以来最差的面试问题：“Oracle 8i中的数据类型varchar和varchar2有什么区别？〃 这是一个可怕的问题。掌握这种琐碎的技术细节和Fog Creek公司想录取你之间没有任何联系。谁会去记这种东西？如果有网络收寻引擎的帮助，你可以在15秒内找到答案。

实际上，还有更差的问题，等会儿我会谈到的。

现在我们要谈到有趣的部分了：面试时提哪些问题。我的面试问题题库清单来自于我去微软公司找第一份工作的经历。这里实际上有几百个微软面试问题。每个人都有偏爱的问题。你也可以发展一套自己的面试问题以及面试的个人风格，这样你就可以比较容易地作出录取/ 不录取的决定。以下是我成功使用过的一些面试技巧， 在面试前，我读一遍应试者的简历，然后在一张纸片上随便写以下我的面试计划。这个计划 实际上就是我要问的问题清单。以下是一个例子（用来面试程序设计师的）：

1. 自我介绍
2. 应试者参加过的项目
3. 无法回答的问题
4. C语言函数
5. 你满意吗？
6. 设计问题
7. 挑战
8. 你还有什么问题？

在面试前，我非常，非常当心，避免自己先入为主。如果在面试前你就已经想当然地认为，一个麻薯工的博士一定是一个有头脑的人。那么在接下来的一小时的面试时间内，无论 那个麻薯工的博士说什么都不能改变你的最初印像。如果在面试前你就认为这个应试者是 个傻瓜，那么他面试时说什么也无济于事。面试就像一个非常精巧的天平。一小时的面试 结束后就要对一个人下结论是不容易的（但是你又必须在面试结束后得出结论）。一些不起 眼的细节可能会影响最后的结论。如果你在面试开始前对于应试者有了一点了解的话，就 好比天平的某一端加上了重重的砝码。这样面试本身就会变得没有用处了。以前有一次在面 试前，一个招聘负责人跑进我的房间说，“你肯定会爱上这个家伙的！”对一个男孩？天 哪，这简直让我发疯。我本来应该说，“嗯，如果你这么确定我会喜欢他，为什么你不干 脆录取他，何必让我浪费时间来面试？〃 但是那时我还太年轻幼稚，所以还是面试了那个 人。当这个家伙开始说一些蠢话时，我对自己说，“哇塞，这应该是个例外情况，也许是 大智若愚。〃 我开始带着有色眼光看他了。于是我以说“录取〃 结束了面试，虽然他 是一个糟糕的面试者。接下来发生了什么事？除了我，其他的面试官都说，不要录取这个人。 教训是，不要听别的人的话，在面试应试者前不要四处打探这个面试者的情况。最重要的是 不要和别的面考官谈论应试者，除非你们都已经作出了独立的判断。这才是科学的做法。

作为面试步骤的第一步，介绍的目的是让应试者放轻松。我通常花30秒钟，讲一下我是谁，接下来面试会如何进行。我总是使得应试者确信，我们关心的是他（她）如何解决问题的，

而不是他（她）的最终答案是对还是错。顺便说一下，面试时，你不要和应试者隔着一个桌子坐着，否则在你和面试者之间就有了一个障碍，并且暗示着一种比较正式严肃的气氛， 这样应试者就很难放松了。更好的办法是把桌子背对着墙壁，或者和应试者坐在桌子的同一边，这样有助于应试者放松。只有应试者不会因为紧张而表现失常，你才能更有效的进行 面试.

第二步的内容就是问应试者最近做了些什么专案。对刚毕业的学生，如果有论文就问问论文，没有的话，就问问他们做过什么很喜欢的大作业.例如，有时候我会问一下，“ 你最喜欢 上学期哪门课程？不一定要和计算机相关的。事实上，如果应试者回答的课程和计算机 没有关系，我会比较高兴。有时候你会发现这个计算机系应届生选择了尽可能少的计算机 相关课程，但是却选修了很多和音乐相关的课程。但是他（她）却说最喜欢的课程是《面向 对象数据库》。哼哼，不错啊.不过如果你直接承认你喜欢音乐胜于计算机，而不是在这儿 胡说八道的话，我会更高兴一点。

当面试有工作经验的人时，你可以让他们谈一下前一份工作。

我问这个问题的目的是在寻找一样质量：热情。在应试者谈到他（她）最近做过的项目时，你观察到以下迹像都是不错的：

- 谈到他们做过的项目时变得热情洋溢；他们的语速更快，语言更生动活泼。这说明 他们对某些东西有兴趣，有热情（因为现实中有许多人对所做的项目根本漠不关心 呢）。即使他们激动地表达对做过的项目的负面感情，这也是一个好的信号。“我 曾经为上一个老板安装Foo Bar Mark II，但他是个傻瓜！ ”表现出热情的人就是我 们要录取的人。差的应试者对工作根本就不关心，所以根本不会激动。一个非常好 的信号是当应试者很激动地谈论上一份工作，以至于暂时忘记了他们正在被面试。 有时候应试者刚开始面试时表现的很紧张–这是很正常的现象，所以我通常忽略 不计。但是当他们谈到单色计算艺术（Computational Monochromatic Art)时，这 个家伙变得极端兴奋，一点都不紧张了。不错，我喜欢这样的应试者，因为他们关 心他们做的事。（什么是单色计算艺术？拔掉你的计算机显示器的电源就可以看到 了）
- 能认真地去解释事情。某些人被我拒掉的原因就是他们不会用普通人能明白的语言 去解释他们做过的项目。很多工科专业的人总是以为所有人都知道Bates理论（译者 注：Bates Theorem，一种经济学的理论）或者Peano公理组（译者注：Peano’s Axioms，数论中的一些定理）是什么。如果应试者开始满口行话了，让他们停一停，然后你说，“能帮我个忙吗？就是为了再练习一下，你能把刚才说的用我老祖母也能理解 的话说一遍吗？”但即便如此，有些人还是继续用那些术语，而且根本没法让人明 白他们在说什么。天哪！
- 如果这个项目是一个团队项目，看看他们是否在有承担领导责任的迹象？ 一个应试 者可能会说：“我们用的是X方法，但是老板说应该是Y，而客户说应该是Z。”我会 问，“那么你怎么做的？ ”一个好的回答可能是“我设法和团队中别的人开了个会， 然后一起搞出个办法…”坏的回答看起来像，“嗯，我什么也不能做。这样的问题 我解决不了。”记住，聪明并且能完成工作。要搞清楚某人是否能完成工作的一个 办法就是看看他（她）过去是否倾向于完成任务。事实上，你可以主动要求他们给 你个例子证明他们能担任领导作用，完成任务。一例如克服公司的陈规陋习。

现在我们谈谈清单上的第三款，无法回答的问题。这很有趣。这个主意的关键在于问一些不可能有答案的问题，就是想看一下应试者怎么办。“西雅图有多少眼科医生？〃 ”华 盛顿纪念碑有多重？ ”“洛杉机有多少加油站？〃 “纽约有多少钢琴调音 师？ “

- 聪明的应试者猜到你不是要测验他们的专业知识，他们会积极地给出一个估计。“嗯， 洛杉机的人口是七百万；每个人平均拥有2.5辆轿车…”当然如果他们的估计完全 错误了也没有关系。重要的是他们能积极地试着回答问题。他们可能会试着搞清楚 每个加油站的储量。“嗯，需要四分钟给一个储油罐加满油，一个加油站有十个油 泵每天运行十八个小时…”他们也可能试着从占地面积来估计。有时，他们的想法 的创造力让你吃惊.而有时，他们直接要一个LA的黄页去查。这都是好迹像。
- 不聪明的应试者则被难住了。他们目瞪口呆地望着你，好像你来自火星。你不得不 提示嗯，如果你想建立一个像洛杉机那么大的城市，你需要建立多少个加油站？〃 你还可以提示他们：“加满一个储油罐要多长时间？”不过，这些榆木疙瘩脑袋还 是只会坐在那里发呆，你得拖着他们往前走才行。这类人不会解决问题，我们可不 要这样的人。

关于编程问题，我通常要求应试者用C语言写一些小函数。以下是我通常会出的题目：

1. 将一个字符串逆序
2. 将一个链表（linked list)逆序
3. 计算一个字节（byte)里有多少bit被设成on
4. 搜索给定的字节（byte)
5. 在一个字符串中找到可能的最长的子字符串，该字符串是由同一字符组成的
6. 字符串转换成整数
7. 整数转换成字符串（这个问题很不错，因为应试者要用到堆栈或者strev函数）

注意，通常你不会希望他们写的程序代码多于5行，因为你没有时间理解太长的程序代码。 现在我们来详细看一看其中几个问题：第一个问题：逆序一个字符串。我这辈子还没有见过 那个面试者能把这题目一次做对。所有的应试者都试图动态生成缓冲区，然后将逆序的字符 串输出到该缓冲区中。问题的关键在于，谁负责分配这个缓冲区？谁又负责释放那个缓冲区？通过这个问题，我发现了一个有趣的事实，就是大多数认为自己懂C的人实际上不理解指针和内存的概念。他们就是不明白。这真叫人吃惊，无法想象这种人也能做程序设计师。但他们真的就是！这个问题可以从多个角度判断应试者：

- 他们的函数运行快吗？看一下他们多少此调用了 strlen函数。我曾经看到应试者写 的strrev的算法竟然只有O(nA2)的效率，而标准的算法效率应该是O(n)，效率如此 底下的原因是因为他们在循环中一次又一次调用strlen函数。
- 他们使用指标运算吗（译者按：原文为pointer arithmetic,指的是加减指针变量 的值）？使用指标运算是个好现象。许多所谓的“C程序设计师”竟然不知道如何使 用指标运算（pointer arithmetic)。当然，我在前文说过我不会因为应试者不掌 握一种特定的技巧而拒绝他。但是，理解C语言中的指针不是一种技巧，而是一种与 生俱来的倾向。每年一所大学要招进200多个计算机系的新生，所有这些小孩子4岁 就开始用BASIC语言在Atari 800s写冒险游戏了。在大学里他们还学Pascal语言，学 得也很棒。直到有一天他们的教授讲了指标的概念，突然，他们开始搞不懂了。他 们就是不能再理解C语言中的任何东西了。于是90%的计算机系学生转系去学政治学。 为了挽回面子，他们告诉朋友，他们之所以转系是因为他们计算机系英俊貌美的异 性太少。许多人注定脑子里就没有理解指标的那根弦。所以说理解指标是一种与生 俱来的质量，而不是一种单纯的技巧。理解指标需要脑子转好几个弯，某些人天生 不擅长转这几个弯。

第三个问题可以考考面试者对C的位运算的掌握，但这是一种技巧，不是一种质量，所以你可以帮助他们。有趣的等他们建立了一个子函数用来计算byte中为1的位的数目，然后你要 求他们优化这个子函数，尽量加快这个函数的运行速度。聪明的应试者会使用查表算法（毕 竟这个表只有256个元素，用不了多少内存），整个表只需要建立一次。跟聪明的应试者讨 论一下提高时间/空间效率的不同策略是十分有意思的事情.进一步告诉他们你不想在程 序启动时初始化查询表。聪明的面试者可能会建议使用缓冲机制，对于一个特定的byte，只 有在第一次被查询时进行计算，然后计算结果会被放入查询表。这样以后再被查询时直接查 表就行了。而特别特别聪明的面试这会尝试有没有建立查询表的快捷方式，如一个byte和它 的置1的bit数之间有没有规律可循？

当你观察应试者写C程序代码时，以下一些技巧会对你有帮助：

- 事先向应试者说明，你完全理解，没有一个好的编辑器光在纸上写程序代码是困难 的，所以你不在乎他们手写的程序代码是否看上去不整洁。你也完全明白没有好的 编译程序和调试器，很难第一次就写出完全没有bug的程序，所以请他们不必为此担心。
- 好程序设计师的标志：好程序设计师写完符号后，通常立刻跟上"}〃符号， 然后再在当中填上程序代码。他们也倾向于使用命名规则，虽然这个规则可能很原 始。如果一个变量用作循环语句的索引，好程序设计师通常使用尽可能少的字符为 它命名。如果他们的循环语句的索引变量的名字是CurrentPagePositionLoopCounter，显而易见他们写程序代码的经验还不够多。偶 尔，你会看到一个C程序设计师写下像if (0==strlen(x))—样的程序代码，常量被 放在==的左边。这是非常好的现象。这说明他因为总是把=和==搞混，己经强迫自己养成这种习惯以避免犯错。
- 好的程序设计师在写程序代码前会订一个计划，特别是当他们的程序代码用到了指 标时。例如，如果你要求逆序一个链表，好程序设计师通常会在纸的一边画上链表 的草图，并表明算法中的索引指针当前移动到的位置。他们不得不这样做。正常人 是不可能不借助草图就开始写一个逆序链表的程序的。差的程序设计师立刻开始写 程序代码。

不可避免的，你会在他们的程序中发现bug，于是我们现在来到了第五个问题：你对程序代码满意吗？你可能想问，"好吧，bug在哪里？〃 这是来自地狱的一针见血的问题，要 回答这个问题可要大费口舌。所有的程序设计师都会犯错误，这不是问题。但他们必须能找 到错误。对于字符串操作的函数，他们通常会忘记在输出缓冲区加上字符串结束符。所有的 函数，他们都会犯off-by-one错误（译者按：指的是某个变量的最大值和最小值可能会和正 常值差1)。他们会忘掉正常的C语句结尾的分号。如果输入是零长度字符串，他们的函数会 运行错误。如果malloc调用失败而他们没有为此写好错误处理程序代码，应用程序会崩溃。 一次就能把所有事情做对的程序设计师非常，非常，非常地少.不过要是真的碰上一个的话， 提问就更有意思了.你说，"还有Bug"。他们会再仔细地检查一遍程序代码。这个时候，观察 一下他们内心是否开始动摇了，只是表面上勉强坚持说程序代码没有问题。总之，在程序设 计师写完程序代码后，问一下他们是否对程序代码满意是个好主意。就像Regis那样问他们！ (译者按，Regis Phi lbin是美国ABC电视网的游戏电视节目主持人，他的口头禅是" 这是 你的最后的答案吗？ ")

第六部分：关于设计的问题。让应试者设计某样东西。Jabe Blumenthal，Excel的原始设计者，喜欢让应试者设计房子。Jabe说，曾经有一个应试者跑到白板前，画了一个方块，这就是他的全部设计。天哪，一个方块！立刻拒绝这样的家伙。你喜欢问什么样的设计问题？

- 好的程序设计师会问更多的信息。房子为谁造的？我们公司的政策是，我们不会录 取那些在设计前不问为谁设计的人。通常，我会很烦恼我得打断他们的设计，说"事 实上，你忘记问这个房子是给谁设计的了。这个房子是给一群长颈鹿造的。〃
- 笨笨的应试者认为设计就像画画，你想画什么就画什么。聪明的应试者明白设计的 过程是一系列艰难的权衡。一个很棒的设计问题是：设计一个放在街角的垃圾箱。 想一想你得做多少权衡，圾箱必须易于清空，但是很难被偷走；易于放进垃圾， 但是碰到狂风大作，里面的垃圾不会被吹出来；垃圾箱必须坚固而便宜。在某些城 市，垃圾箱必须特别

设计，以防恐怖分子在里面藏一个定时炸弹。
- 有创造力的应试者会给出有趣而独特的设计。我最喜欢的问题之一是为盲人设计一 个放调味品的架子（译者按：原文为spice rack老外的厨房里有个专门放调味品 的架子，上面放了很多小罐罐，里面装了各种各样的调料）通常许多应试者的建议 是把布莱叶文（一种盲人使用的文字）刻在放调料的罐子上，这样文字会卷起来而 变形。我碰到一个应试者，他的设计是把调料放在抽屉里，因为他觉得水平地感知 布莱叶文比垂直地做更方便。（试试看！）这个答案这样有创意，使我震惊！我面 试了有一打的程序设计师，从来没有人想到过类似的答案。这样有创意的答案确实 跃过了普通人考虑问题的条条框框。仅仅因为这个答案太有创意了，而且应试者别 的方面还过得去，我录取了这个应试者，他现在已经成为Excel团队中一个优秀的 项目经理了（译者按，本文作者曾在微软工作过）。
- 总是争取一个确定的了结。这也是完成工作的特质的一部分。有时候应试者会犹犹 豫豫不能作出一个决定，试图回避困难的问题，留着困难的问题不作决定就直接向下进行，这很不好。好的应试者有一种推动事情自然地前进的倾向，即使你有意把他们 拖回来。如果关于某个话题的讨论开始原地打转变得没有意义了，好的应试者会说，“嗯，我们可以整天谈论这个，但是我们得做点什么。为什么我们不开始…”

于是我们来到了第七部分，挑战。这部分很好玩。在面试中留心一下，当面试者的回答绝对的百分之百毫无争议时，你可以说：”嗯，等一下等一下.”然后花上两分钟玩一下魔鬼的 游戏（译者按，原文为devil’s advocate,魔鬼代言人指的是违背自己的良知，为错误邪恶 的观点辩护）.记住一定要在你可以肯定他正确时和他争论。

这个很有意思.

- 软弱的应试者会屈服。那我就和他说拜拜了。
- 坚定的应试者会找到一个办法说服你。他们会以甘乃迪总统的口才来说服你，“也 许我误会了你的意思，”他们这样开头，但是正文仍是坚定地站稳立场。这样的人 我就录取。

不得不承认，面试双方的地位并不是平等的。有可能应试者由于害怕你的权力而不敢于你争辩。但是，好的应试者有足够的热情和勇气坚持正确的观点，他们由于热切希望说服你而会暂时忘记正在被面试。这样的人就是我们要录取的人。

最后，可以问一下应试者有什么想问的。一些人喜欢看看应试者这时是否会问一些聪明的问题。这是市面上流行的面试书籍的标准技巧。我个人不在乎应试者问什么，因为这时我已经做了决定。麻烦在于，应试者也许已经见了5、6个人，进行了好几轮面试，他们可能很累了， 以至于不能为每轮面试都准备一个聪明而独特的问题。所以如果他们没有可问的，没关系。

我总是留下面试的最后5分钟来推销我的公司。这很重要。即使我不打算录取眼前这个应试者。如果你幸运的找到一个很棒的应试者，你当然愿意做任何事情说服他（她）来你的公司。 即使他们不是好的应试者，你也要尽力让他们为Fog Creek公司激动，这样面试结束时

他们 会对Fog Creek公司留下一个很好的印象。记住，应试者并不仅仅是可能的雇员，他们也是 顾客，也是我们公司的推销员。如果他们觉得我们的公司很棒，他们也许会推荐朋友来面试。

啊哈，我记得我说过我会给出一些应该避免的非常不好的反面的试题例子。

首先，避免不合法的问题。有关种族，宗教，性别，出生国，年龄，服役记录，是否老兵，性倾向，生理障碍的问题都是不合法的。即使他们的简历说他们1990年在军中服役，也不要问有关问题。也许这会让他们愉快地谈论在海湾战争中的经历。但是你的问题还是不合法的。 如果简历上写着他们上过Technion in Haifa,不要问他们是否是以色列人，即使只是为了闲谈，因为这是违法的.下面有一个很好的不合法的例子点击这里有很多关于什么是违法 的讨论。（但是这个网站的其余问题够愚蠢的。）

其次，不要在问题中给予应试者暗示，我们公司喜欢或者不喜欢什么样的员工。我能想到的一个例子是问应试者是否有小孩或者是否结婚了。应试者也许会想我们不喜欢有家庭拖累的员工。

最后，不要问那些脑筋急转弯的题目，例如6根火柴怎么拼出4个三角形。像这样的灵机一动的问题是不能看出应试者是否有“有头脑/完成工作〃 的质量。

面试与其说是科学不如说是艺术。但是只要你记住有头脑/完成工作这个原则，你就可以应对自如。有机会就问问你的同事他们喜欢的面试问题和答案。这是我们公司员工午饭时热衷的话题之一。

二十一、重金激励害多利少

Mike Murray是一位非常衰的微软人事经理，他做了不少蠢事，不过最经典的还是他上任后 不久推出的「出货」奖，也就是在你负责的产品上市时送你一块压克力做的大墓碑。这应该 算是某种驱使你工作的激励，因为如果你不好好做事就拿不到那块墓碑！真让人奇怪这个墓 碑出来之前微软是怎么推出产品的。

「出货」项目在一场大型的公司聚餐中大张旗鼓地宣布。在活动举行前数星期，公司园区就布满了宣传海报，上面有比尔盖茨的照片还有一句「这个人为何在微笑？」。我不确定那是什么意思。是说因为现在有了能让软件出货的激励方案，比尔觉得高兴而微笑吗？不过在那场聚餐活动上，雇员们显然觉得自己被当成小朋友。到处都是不满的嘘声。Excel的程序团 队还举起一个大告示板，上面写者「Excel团队为什么在打哈欠？」这个出货奖被极其鄙视， 结果连Douglas Coupland的经典作品Microserfs里甚至还有一章，里面有一群程序员尝试 用一根吹管把它破坏掉。这是真有其事，不是我乱盖的。

把你的科学家雇员当作幼儿园小朋友来对待，并不是很少见的特殊现象。几乎每家公司都有同类型既侮辱又贬低人的奖励方案。

在我工作过的两家公司里面，压力最大的时间就是每年两次的绩效评估期。不知怎么回事，Juno和微软的人事部门的的绩效评估系统一定是从同一本呆伯特式管理书籍上抄的，因为两边的作法完全一样。首先你要「匿名」地向上评估你的直属经理(假装这样就能很诚实的进行）。然后再填好一个可有可无的「自我评估」表格，让你的经理在评估你的绩效时可以「参 考」。最后你会在多个不可量化的项目（如团队合作)得到一个1到5的分数(不过实际上只会 出现3分或4分）。经理会把建议奖励名单往上送，接下来名单会被完全忽视，然后每个人都 会拿到完全随机的奖金。这样的系统从未考虑一个事实，就是人各有不同而独特的天赋，需 要各种天赋才能让一个团队好好运作。

有几个原因使得绩效评估令人紧张。我有许多朋友的才华都不是传统尺度可以呈现的，他们通常都拿不到好考绩。举例来说，某位朋友是个欢乐触媒和快活的总监，可以在艰苦状况下激励大家，而且还是结合团队的黏着剂。不过由于经理不了解他的贡献，所以常会得到负面考绩。另一个朋友非常的有洞见；别人会因为跟他讨论事情而把工作做得更好。他会花高于平均的时间去尝试新技术；就这方面来说，他对团队其他人的帮助是无法衡量的。不过他写的程序行数也低于平均水平，而他的经理也实在太蠢没注意到其他贡献，所以通常考绩也不好。而负面的评价显然对士气有致命的作用。事实上即使给某人正面的评价，只要评价不如本人预期的好，对士气一样有负面的作用。 绩效评估对士气的作用并不是对称的：负面评价对士气伤害很大，可是正面评价并不会改善 士气或生产力。拿到正面评价的人都已经做得很好了。对这些人而言，正面评价会让他们觉 得自己是为了拿到好成绩才好好工作的，就像巴伐洛夫制约实验中的狗为奖赏而工作，并非 真心在意自己工作质量的专业人仕。

其中的困难在于大多数人都认为自己把事情做得很好(即使事实上不是）。这其实是我们心里为了让生活不那么难受，而对自己玩的一个小把戏。所以说如果每个人都觉得自己表现良好， 而评估结果正好是公正不阿(这并不太容易做到)时，那么大多数人都会对评估结果感到失望。 而这种结果所牺牲的士气成本很难被低估。在团队中诚实地实施绩效评估通常会导致士气低 落或忧郁，甚至会有部份人员离职，影响长达一星期左右。团队成员间会产生间隙，通常是 因为考绩差的人嫉妒考绩好的，发生如同DeMarco和Liste「称之为团队自杀(teamicide)的 过程：己冻结的团队不自觉的毁灭。

Alfie Kohn在哈佛商业评论中一篇己成为经典的文章中写道：

过去三十年间至少有两打以上的研究明确地指出，为了报酬而工作的人，表现不如完全 不期望有报酬的人。HBR Sept/Oct 93 他的结论是「激励(或者说贿赂)在职场上是行不通的」。DeMarco和Liste「更进一步明白地 表示，任何型式的职场竞争及奖惩方案，包括以前流行那种「在某人做对事时马上奖励」的 把戏，所带来的伤害都大过好处。给某些人正面激励（比如愚蠢的公开颁奖仪式)暗示他们 其实只是为了拿那个压克力奖牌才有表现；也暗示他们在工作上不够独立，要有甜头才会努 力；这实在是既侮辱又贬低人格。

大多数的软件经理都没有选择余地，只能依循现有的绩效评估系统。如果你身在这个位 置，要防止团队自杀的唯一办法，就是对每个人都只给些装装样子的考绩。不过如果你在这 件事上有其他选择，我会建议避开各种的绩效评估和奖励方案，或是愚蠢的本月最佳员工计 划。

二十、二不配备测试人员的五个首要（错误）原因

1992年，James Gleick正和一个多虫软件的各种问题周旋。科学作家Gleick认为 微软刚出的Word for Windows新版很可怕。他在星期天纽约时报杂志写了 一长篇 文章，这篇文章只能以激动来形容，整篇都在指责Word团队不理会客户的要求， 交付问题如此多的产品。

稍后，他以地区 I nternet供货商Pan ix(刚好也是我的I nternet供货商)客户的身 份，要求提供自动排序并过滤其邮件的方法。提供这个功能的是一个叫procmail 很神秘的UN IX工具，它的接口连最硬派的UN IX迷都觉得难用。

总之Gleick先生不小心在procmail里打错字，结果把他所有的信都删掉了。一气 之下就成立自己的Internet联机公司。他雇用了 Uday Ivatury当程序员，并且 建立了 Pipeline这个真的有点超越时代的公司：这是第一个有图形接口的商业 Internet联机供货商。

当然啰，Pipeline也有它自己的问题。很早期的第一个版本没有使用任何错误更 正的协议，所以很容易把数据弄乱或当掉。它就像所有的软体一样有臭虫。1993 年我去Pipeline找工作，面试中问到Gleick所写的文章。「现在你已经身在萬笆 的另一边，」我问道：「有没有稍微体会到创造好软件的难处呢？」

Gleick恼羞成怒。他否认Pipeline有任何问题，他也否认这个软件有和Word—样糟。他告诉我说：「约耳，总有一天你也会恨微软的。」他做了一年的软件开 发而不再只是个使用者，这一年的经验却没有让他体会到无虫又易用的软件是多 么困难的事，这实在令我有点震惊。所以我就回绝那个工作机会逃掉了。 (Pipeline后来被一家世界上最奇怪的Internet供货商PSI买走，然后随随便便就 被关掉了。)

软件有虫。CPU异常的守规矩，它们绝对会/互绝处理没有被明确教过的东西，而 且通常会以最孩子气的方式拒绝。我的笔记本电脑在外头使用时常会当掉，因为 它找不到本来找得到的打印机，多么像个婴孩。不过原因可能是某一行程序有个 极微小却又很严重的问题。

这也正是你绝对要有QA部门的原因。每用两个程序员就需要一个测试人员（如果 软件需要用于多种复杂的组态或操作系统时还不只）。每个程序员都应该和某一 个测试人员紧密配合，尽可能频繁地丢出自己编译的版本给测试人员测。

QA部门应该要独立并拥有权力，绝对不可以对开发团队负责，事实上QA的主管应 该有否绝权，可以阻止发行不合格的软件。

我第一份真正的工作是在微软；这家公司并不以高质量的程序傾名，不过它还是 雇用了大量的测试人员。所以我曾经认为每个软件组织都会有测试人员。

虽然很多组织确实有，不过没有测试人员的组织还是多的令人惊讶。事实上很多 软件团队甚至不相信测试。 你可能会认为，现在的经理经过80年代的质量狂热洗礼，又有各种无意义的国际 「质量」认证(如IS0-9000)和「六个标准偏差」等术语，应该会了解高质量的 产品才能

获得高质量的生意。事实上他们都了解，其中大多数人已经尽量把它塞 进脑袋，不过他们还是找出各种理由不用软件测试人员，而这所有的理由通通 都是错的。

我希望可以对你解释这些想法的错误。如果你赶时间剩下的就别看了，直接算人 头每两个全职程序员请一个全职测试人员就对了。

下面是我听过不雇用测试人员最常见的鬼叫：

1.问题是懒惰的程序员弄出来的。

这种幻想的内容是这样的：「如果你请了测试人员，程序员就会马虎而写出问题更多的程序。 不要请测试人员，就可以强迫程序员一开始就写出正确的程序。」哇！如果你会这样想， 那你不是没写过程序，就是把写程序这回事完全想歪了。由定义来看，臭虫会出现就是因为 程序员没看到程序代码里的问题。通常就是需要其他人才看得到问题。当我还在Juno写程 序时，每次都会用一样的方式执行我的程序。我会照我自己的习惯大量使用鼠标操作。我们 神奇的测试强者习惯有点不同：她比较常用键盘(而且会严厉地实际用所有可能的组合去测试使用接口）。这样子很快就会找出丈量的问题。事实上有时候她还会说整个接口都不能用， 而且是沒分之百不廉用，可是在我的机器上一切正常。当我看着她重现问题才恍然大悟。Alt 键！你按住Alt键了！我怎么会没测到呢？

2.我的软件放在网络上。即使有问题也马上就能修好。

哈哈哈哈！好吧，你说对了，用网络分发软件的确能比以往软件包时代更快速的更新修正版本。不过即使把软件放在网站上，也别低估在项目冻结后修正一个问题的代价。一来你可 能在修正第一个问题时产生更多的问题。更糟的是如果你检讨发行新版本的程序，就会了解 在网络上丢出修正版是个相当昂贵的建议。更别提会让人因为有以下感觉而造成坏印象了：

3.客户会替我测试软件。

啊，令人害怕的「Netscape防线」。这家可怜的公司借着独特的「测试」方法严重伤害了自己的声誉。他们的测试方法是：

1. 当程序员写好一半时，不经任何测试就把软件发行到网络上。
2. 当程序员说完成时，不经任何测试就把软件发行到网络上。
3. 重复做六到七次。
4. 把其中某个版本称为「最终版」
5. 每次当c|net上爆出一个丢脸的问题时就发行.01、.02、.03版。

这家公司是「广泛beta」这个点子的先驱。意思是让教百?万人下载这些多虫的未完成版本。最初几年几乎所有Netscape使用者都在用某个抢鲜版或beta版。结果大多数人都认为 Netscape软件的问题实在很多。虽然最终版通常问题都极少，可是Netscape已经让多到离谱的人用过多虫的版本，所以大多数人对该软件的普体印象都很差。

另外让「客户」来测试的重点在于让他们找到问题然后由你来修正。不幸的是，Netscape 或任何其他地球上的公司都没有人力，可以处理两百万客户的问题回报并找出真正重要的 问题。当我要填报Netscape 2.0的问题时，问题回报网站一直当机，根本不让我回报(不过 反正问题回报最后都会丢进黑洞）。可是Netscape并没有学到教训，现在6.0「抢鲜版」的 测试人员就在新闻组里抱怨，说问题回报网站还是没办法送出报告。那么多年了！还是老问 题！

不过我敢打赌，在这些多得要命的问题回报中，绝大多数都在讲相同的5到10组很锻显的问题。里面可能还埋有一两个有意思又难查的问题，是某人千辛万苦才成功送出的。不过反正没人会看这些报告，所以也就等于不见了。

这种测试型式最糟的一点，就是公司会得到非常差的印象。当Userland推出主力产品Frontie「的第一个Windows版时，我去下载并开始照着教学操作。不幸的是Frontie「当了好几次。我完全照着教学上面的指令逐一照做，怎么样都没办法执行两分钟以上。我觉得Userland可能连最基本的测试都没做，根本没有确认教学是能动的。产品质量低落的印象让我有很长一段时间不再碰Frontier。

4.有资格可以胜任的人都不想做测试人员。

这是个很沈痛的理由。好的测试人员非常难找。测试人员和程序员一样，顶尖高手和一般 水平的人相差一个量级。Juno有一个测试人员Jil l McFar lane，她找到的问题是其他四个人 合起来的三倍。我是真的算过，绝对没有夸大其辞。她比普通测试人员的生产力好12倍。当 她离职时，我写了封电子邮件给执行长说道：「我宁愿只有星期一和星期二让Jill帮忙，也 好过让QA团队其他人全部一起测。」不幸的是，这么聪明的人大都容易对日复一日的测试 感觉厌烦，所以顶尖的测试人员通常只待三或四个月就会换工作。这个问题唯一的解法就 是面对然后想办法处理。下面列出几个建议：

把测试当作技术支持的下一个晋升工作。测试或许很沈闷，不过绝对比用电话处理愤怒的用户好，而且可能也是消除某些技术支持人员骚扰的方法。

允许测试人员上程序设计课程拓展职业生涯，并且鼓励较聪明的测试人员，运用程序工具及脚本语言开发自动化测试套件。这比不断重复测试相同的对话盒要有趣多了。

要有顶尖测试人员经常流动的认知。积极的进行招募以保持稳定的人员流入。不要只因为暂时满额而停止招募，因为总会有下一波好景气。

找寻「非传统」的工作者：找聪明的青少年，大学里的小朋友还有退休人士来兼差。你可以用两三位顶尖全职人员再搭配一群趁暑假赚学费的Bronx Science(纽约一流高中）学生，建立出一个效果惊人的测试部门。

雇用临时人力。如果请10个临时工进来拿你的软件来玩个几天，就会发现很多很多的问题。里头可能会找到两三个测试技巧不错，值得签约转成全职人员的人。要先有心理准备，里面有些人可能做不了测试人员；这时只能请他回家再继续找新人。这也正是临时人力中介的功用。

下面这种方法无法处理这个问题：

千万不要对来工作的计算机科系学生打歪主意，说什么「所有人都要在QA部门磨练一阵子才能去写程序。」我看过太多例子了。程序员没法子当个好的测试人员，而且还会因而损失好的程序员，而这就更难找了。

终于到了，这是人们不雇用测试人员的头号笨理由：

5.我请不起测试人员！

这是最笨而且也是最容易揭穿的理由。不管测试人员有多难找，还是比程序员要便宜，而 且便宜多了。另外如果不请测试人员，就得让程序员自己测。再来如果你认为测试人员大量 增加不太好，就想想这件事的代价吧：年薪十万美元的明星程序员受不了有人要他「在发 行前留几个星期测一测」，而换去比较专业的公司。光是为找新程序员而付给猎人头公司 就够你一年请三个测试人员了。吝啬请测试人员是如此的划不来，我实在不懂为什么有这 么多人认不清。

二十三、任务换人有害无益

在管理一个程序团队时，第一件要学的事就是任务配置(task allocation)要正 确。「任务配置」只是把事倩分文家微的夸大说法。用希伯来文的普通话来 说就是「倒档案」（因为你会把档案倒在某人身上）。有些事情做得对会得到不可 思议的生产力利益，决定哪些档案要倒在谁身上就是其中之一。反过来没做好 的话可能就会陷入麻烦的状况，没有人能做好何任何事情而且大家都抱怨「在这 里什么事都做不起来。」 由于这是个针对程序员的网站，我要拿个程序设计问题让你的脑袋动一动暖暖身。

假设你要做A和B两件运算要做。每一件都需10秒的CPU时间。现在你有一颗CPU， 为了简化问题，所以工作序列中没有其他东西。

在我们的CPU中可以选择是否用多任务处理。所以你可以先做好一件再做另一件。

循序处理



也可以使用多任务方式。如果用多任务的话可以假设这颗特别的CPU每个工作每 次可以执行一秒，而且工作切换完全不花时间。

多任务处理



你会选哪一种方式呢？大部份人的直觉反应都认为多任务比较好。不管哪一种状 况，都得等20秒才能两件运算都完成。不过可以想想单就各洋运算来说要多久才 有结果。

在两种状况下，运算B(标成蓝色)都要20秒才得到结果。不过运算A的结果在多任 务时需要19秒。可是循序时就只要10秒就好了。

换句话来说在这个安排好的例子中，循序处理的每洋运算游f均对/琢比多任务处 理少(15秒对19.5秒）。（事实上这例子也并不是真的那么假–它是源于Jared在工作中必须解决的一个真实问题）

方法	运算A花的时间	运算B花的时间	平均
循序处理	10秒	20秒	15秒
多任务处理	19秒	20秒	19.5秒

我刚刚说过「工作切换完全不花时间」。其实在真的CPU中工作切换是需要一点 点时间的，基本上要足够储存CPU缓存器的状态并加载其他工作的CPU缓存器。实 际上这短到几乎可以忽略。不过为了让生活更多乐趣，让我们假设工作切换需要 半秒。现在情况变得更糟了：

方法	运算**A**花的时间	运算**B**花的时间	平均
循序处理	10秒	20秒 + 1次工作切换 = 20秒	15.25秒
多任务处理	19秒 + 18次工作切换 = 28秒	20秒 + 19次工作切换 = 29.5秒	28.75秒

现在呢，虽然我知道这有点蠢，不过就算为了让我高兴一下，想想如果工作切换需要一分钟那如何？

方法	运算**A**花的时间	运算**B**花的时间	平均
循序处理	10秒	20秒 + 1次工作切换 = 80秒	45秒
多任务处理	19秒 + 18次工作切换 = 1099秒	20秒 + 19次工作切换 = 1060秒	几近19分钟！！

工作切换用的时间愈长，多任务处理的代价愈大。

这件事本身不怎么新奇，不是吗？不久大概就会有些白痴气愤地写信指控我「反 对」多任务处理了。他们会质问我：「你真的想要回到那种得先结束WordPerfect 才能执行Lotus 1-2-3的DOS时代吗？」

不过那并不是我的意思。我只是想要你同意，在这类例子中：

a)循序处理会让结果次上比较快得到，而且 b)工作切换需要愈久，多任务处理所付的代价就愈大。

够了，别管CPU 了，来管管人吧，这有趣多了。这里的重点在于管理「程序员」 时，工作切换会需要很长很长的时间。因为程序设计这种工作必须同时在脑袋 里记很多东西。另外记住的东西愈多，写程序时生产力愈高。用全速写程序的程 序员脑里随时都会记住无数的事情：变量名称，数据结构，重要的API，写过常要 用到的辅助函数名称，甚至存放原始码的次目录名称，一切东西都要记住。如果 你把程序员送到克利特岛去度假三星期，他所存东西通通都会忘掉。人脑似乎会 把东西移出短期RAM，改存到永远都读不回来的备份磁带上。

要多久呢？嗯，我的软件公司最近放下手头上在做的事(开发一套代号CityDesk 的软件产品），花了三星期去帮助某个客户处理一个紧急状况。当我们回到办公 室时，感觉好像要另夕A三星期才能回复全速制作CityDesk。

就个人层次来说，你曾经注意过某件事吗？叫某人做一个工作可以做得很好，可 是如果给他两个工作，他会把其中一个做好却忽略另一个，不然就是两件工作都 做得很慢，慢到你觉得#龙蜜都比他勤劳。这是因为程序设计的工作就是需要很长 的切换时间。就我自己来说，当我需要同时完成两个程序设计项目时，切换时间 大概要六个小时。以一天八小时来看，等于说多任务处理把我的生产力降到每天 只剩二小时。真令人沮丧啊。

同样的道理，如果你给某人两件工作，应该要感谢他们只做一件工作而放弃 另一件，因为这样能做好更多的事，而且平均上也能更快完成工作。事实上这一 切的重点就是绝对不要让人同时做一件以上的事。请确定你有明白它的意思。好 的经理人会认为自己的责任是濟餘摩涛，好让大家都能专注在一#事倩并把它真 的完成。遇到紧急状况时，请先想想能不能自己处理掉，真的不行再丢给深陷在 专案中的程序员吧。

二十四、绝不去做的事情，第一部

Netscape 6.0第一个公开的beta版终于出来了。5.0版从来没出现过，最后一版重大改版是 大约三年前发行的4.0版。在Internet世界里三年的时间长得可怕。就在这段时间中， Netscape只能无望地坐看市场占有率直线下降。

我这样批评他们在两个版本间等了很久，似乎有点惹人厌。他们并不是故意的这样的吧。真有人会故意这样做吗？

呃，没错，他们是故意的。他们做了一个每家软件公司都可能犯的一个最糟的策略错误: 他们决定把程序从头重写过。

Netscape并不是第一家犯这种错的公司。Borland买下Arago，想把它变成Windows版的dBase 时就犯了同样的错。这个注定失败的项目用了很长的时间开发，长到让它被微软的Access 彻底打败。Bor land后来又重蹈覆辙整个重写QuattroPro,还想拿重写的新功能来吓人。微 软几乎也做了相同的蠢事，想要在一个叫Pyramid的专案里重写Windows版的Word,不过这个 案子已经结束，并且消失在尘土之间。微软很幸运，他们从未停止使用原有的程序代码库， 所以一直有东西可以推出。因此这个错误只成为财务上的问题，而不是策略上的灾难。

我们都是程序员。而程序员在内心其实是个设计者，他们每到一个地方第一件事，就是先把一切铲平然后创造伟大的事物。我们对修补改进和种种花床的渐进式革新不会感觉兴奋。

程序员总想把旧程序丢掉重新开始，其中的原因很微妙。他们会认为旧的程序是一团乱，不过下面这有趣的观察指出他们可能是错的。他们会认为旧程序一团乱的直正原因是一个很基本的程序设计原理：

读程序比写程序困难。

这就是程序再使用如此困难的原因。这也说明为何团队中每个人都喜欢用不同的函数把字符串切成字符串数组。他们自己写自己的函数，因为这样比去弄懂旧程序更简单而有趣。

依据这个原则推论，你可以询问任何一个现役程序员正在用的程序代码。几乎每一个人都会告诉你：「这真是一团乱，我真想把它丢掉重新开始。」

为什么会说是一团乱呢？

「嗯，」他们说：「看看这个函数。有两页那么长耶！这些东西都不属于这里的！我真不知道里头一半的API是叫来做啥的。」 在Borland新的Windows版电子表格软件推出前，该公司有意思的创办人Philippe Kahnwas 在媒体上到处吹墟，说些QuattroPro是整个重写的，所以比微软Excel好上多少倍等等的鬼 话。全新的原始码哦！好像原始码会生锈一样。

新程序代码比旧程序代码好的想法显然是很荒唐的。旧程序代码已经被用过也被测试过。很多问题都已被找出来并被修好。它并没有什么问题，不会因为在你的硬盘里放久了就生出新的问题。你错得离谱了，宝贝！软件应该要像旧奔驰一样，光在放在车库里就会生锈吗？软件应该要像泰迪熊那样，不用全新的材料做就不好吗？

回到那个两页长的函数吧。是的，我了解这只是个显示某个窗口的简单函数，不过却多了很多怪东西而且没有人知道为什么。嗯，我会告诉你原因：它们是问题的修正。其中一个是南西在没有Internet Explore「的计算机上安装时写的。另一个是对付内存不足时的问题。还 有一个是要应付档案放在软磁上，用户却中途抽掉磁盘的状况。LoadLibrary呼叫很难看， 不过却能让程序在旧版Windows 95上也可以执行。

里面每个问题都是在真实世界使用好几个星期之后才发现的。而程序员可能也花了好几天才在实验室重现并予以修正。问题看起来很多，可是解决方法可能只有一行程序，甚至可能只有几个字符，可是光两个字符的背后有着许多的时间和心血。

当你把程序代码丢掉从头重写时，其实是把这所有的知识都丢掉了。这所有已修正的问题，好几年的写程序的成果。

你会让出市场领导地位。你会把两三年的时间当礼物送给竞争者。相信我，软件的一年可是很长的。

你会让自己陷入一个极端危险的位置，到时候会有很多年都只能发行旧版程序，完全无法因应市场需求的新功能而改变策略，因为你没有可以发行的程序。这段时期最好还是把公司收起来算了。 你会浪费大量的资金去撰写已有的程序。

有代替的方案吗？大家都同意Netscape的旧程序似乎真的非常糟。嗯，或许它真的很糟，不过你知道吗？它还是能在世界上数不清的计算机系统上运作的还不错。

当程序员照他们一贯的作风，说他们的程序是团神圣的垃圾时，其实里头会包含三种错误的理由。

首先是架构上的问题。程序代码并没有被正确地安排。网络联机的程序代码会在莫名奇妙的地方自己显示对话盒；而这应该是由使用接口来处理才对。你可以小心的搬动程序代码，重构并修改接口，一次一个地解决这些问题。你可以找一个程序员小心处理好，再把所有修改一次放回版本管理系统，就不会影响到其他人。即使很重大的架构修改，都不必丢掉程序代码就可完成。我们花了好几个月重整Juno项目的架构：只是把东西搬来搬去，把程序代码清理干净，建立合理的基脆别，并且在模块间建立明确的接口。不过我们非常小心的处理现 有的程序代码库，结果没有产生新问题也不必把还能用的程序丢掉。

程序员认为程序代码一团糟的第二个原因是效率不好。谣传Netscape的成像程序很慢。不过这只牵涉整个项目的一小部份，你可以优化甚至重写这一小部份，并不需要把所有程序都重写。在优化速度时，抓到1%的工作重点可以得到99%的效果。

第三个理由是说程序代码他X的丑。某个我做过的案子真的有一个叫FuckedString的数据类型。另一个案子则是在开始时在成员变量名字前加底线1’，后来却改用比较标准的*1m*’。结果一半的函数是用开头而另一半是用1m_’开头，看起来很丑。坦白地说，这种问题用 Emacs的宏功能五分钟就解决了，不需要从头开始。

你一定要记住，在要从头重新开始时，完全没有理由相信这次会做得比第一次好。首先你的程序团队根本不可能和当初相同，所以并不会真的有「更多的经验」。你其实只会把大部份

的旧错重新再犯一次，另外再多加一些旧版本没有的新问题。

在面对大型商业应用程序时，做了就丢掉是很危险的旧咒语。如果你只是写程序做做实 验，想到更好的算法时大可把上星期写的函数丢掉，这是很正常的。想藉重构让某个类别更 容易使用，这也不会有任何问题，不过把整个程序都丢掉是个危险的愚行。如果Netscape 真的运用软件业经验成熟地管理，就不会把自己害得这么惨了。

二十五、冰川下的秘密

「我不知道我的开发团队怎么了，」执行长在心里对自己说：「刚开案时情况还 很顺利的。整个团队像疯了一样，前几个星期就做出一个很好的可用原型。可是 从那之后事情似乎慢得像在爬一样。他们根本就是在偷懒吧。」他选了一根 Callaway钛制一号木杆，叫球僮去弄杯冰凉的梓檬水。「或许我应该开除几个表 现最差的，这样应该能让他们动作快一点！」

这时候开发团队当然不支^道有什么不对。事实上没什么不对，他们完全按照进度 进行。

别让这种事发生在你身上！我要告诉你一个小秘密，是关于那些非技术的管理人 员的秘密，可以让你的日子好过一百万倍。这其实很简单。知道了我的秘密之后， 你和非技术背景的经理合作时再也不会有麻烦（除非你跟他争论他的高尔夫球杆 的反弹系数）。

很显然的程序员用某种语言思考，而MBA则是用另一种语言思考。我曾经对软件 管理中的沟通问题思考过好一阵子，因为对我来说，能在程序员和MBA间翻译沟 通的人少之有少，一旦掌握权力和报酬显然都会随之而来。

由于我是由软件业开始工作的，几乎我做过的所有软件都是所谓的「猜测 (speculative)」软件。意思是该软件并不是为特定客户而制作，而是依据浪多 浪多人会买的期望而建立的。不过很多软件开发者并没有这种福气。他们可能是 在当顾问，要为单一客户开发一个项目，也可能是企业内部的程序员，帮公司写 个会计用的东西(或者其他内部程序员会写的东西，反正对我来说很神秘）。

你是否曾经注意到，在这些客制化的案子中如果出现进度延误、失败或各种灾难， 都有一个最常见的原因。这个原由基本上可以浓缩下面这句话：「客户（应该说 多余的客户）并不知道他们要的是什么？」

这边列出同一种症状有三种版本:

1. 「这混账客户不停的改变主意。一开始要主从式架构。然后又在 Delta航空的空中杂志读到XML，结果又决定要加XML。现在为了配 合整群的乐高Mindstorm小机器人，我们又要整个重写。」
2. 「我们完全.帶餚//7游要求制作软件。合约中把最小的细节都写清楚。 我们也照合约内容交付软件。结果等我们交付时他们却垂头丧气 的。」
3. 「我们的烂业务同意了一份爾定价游合约，答应要写基本上未明确定义的东西，而客户的律师又够精，抓紧合约里的条款说除非客户 接受，否则不用付钱，所以我们只好投入9个人的团队去做他们的 案子，做了两年才只拿到800美元。」

如果有某件事很重要，必须用2500转大电钻注入每个生手顾问的脑袋里。那就是下面这个件事：客户不知道他们要什么。别再期望客户知道他们要什么。这种事就是从来没发生过，你就认了吧。

代替的作法是假设你友正都得做出一些东西，而客户也得喜欢你做的东西，最多 只是会有点惊讶而已。你得自己去研究，自己去找出一个设计，能皆大欢喜地解 决客户的问题。

你要设身处地替他们着想。想象你刚把公司用一亿元卖给了雅虎，决定要翻修厨 房就找了专业设计师来。你给他的指示只有「要和威尔和葛蕾丝（Will and Grace, 译注：美国喜剧)里的厨房一样酷。」你根本不知道要怎么做到，也不知道自己 想要一个维京炉具和Subzero冰箱，这些都不是你会的字。你只要设计师做出一 些好东西，这就是你请他的原因。

搞极限程序设计的人会说，解决的方法就是把客户带进房/琢，把他们当成开发团 队成员一样参与各个设计程序。我觉得这也太「极限」了一点吧。这就像是设计 师在设计厨房时要我在场参与，然后要我对各个小细节表示意见。对我来说这实 在很无聊，如果我想当个设计师我自己去当就好了。

不管如何，反正你不会真的要客户待在你的团队里，是吧？客户代表很可能只是 会计部门抓来的某个小人物，派去陪程序员作业只是因为他是部门里做得最慢的， 所以没有他做事也没差。接下来你只好牺牲原本的设计时间，用最简单明了的方 式全部解释一遍。

假设你的客户不知道他们要什么，然后依照你对该应用领域的了解自己去设计。 如果你需要花时间去学习该应用领域的知识，或者需要该领域的专家帮忙，这都 是正常的。不过要记得软件的设计才是你的工作。如果你能搞懂该应用领域而设 计出一套好的使用接口，客户就会很高兴。

回到主题，我答应过要告诉你一个秘密，那个关于软件的客户（或非技术经理）以及程序员两者间语言翻译的秘密。

你知道冰山有90%是在水面下吗？没错，大部份的软件也是这样。那些漂亮的使 用接口只占10%的工作，而其他90%的程序设计都是看不到的。如果再考虑到一半 时间在抓虫这个事实，使用接口就只占了 5%的工作。如果只计算使用接口中的视 觉部份(能在PowerPoint里看到的部份），其实就不到1%了。

这并不是秘密。真正的秘密是非程序人员不知遣这件事。

这个冰山秘密有些很重要的推论。

重要推论一：把使用接口的画面展示给非程序人员看时，如果这个接口很不好， 对方会认为你整个程序也是很不好的。

我在当顾问时学到这一课，当时我对某客户的执行团队展示一个重要的web项目。 项目本身的程序几乎都已完成，不过还在等美术设计师选择字型和色彩还有漂亮 旳立体活页图案。等待期间还是用普通字型和黑白色，画面上有很多地方都空白 而不太美观，基本上就是不好看啦。不过功能全部都在，而且有些很不错的成果。

猜猜展示的结果如何？客户磨个会议都在紧盯着画面的美术外观。他们连使用接 口都没提，只管图形好看不好看。他们的项目经理抱怨着说：「这着起来,就是不 太#V落。7他们能想到的就只有这些，我们没法子让他们考虑到功能。图形设计 显然不用一天就可以修好，而整个

感觉就像他们认为自己找的是厘家

重要的推论二：把使用接口的画面展示给非程序人员看时，如果这个接口非常漂 亮，对方会认为这个程序几乎已经完工。

非程序人员只会看着画面，他们看到的是一堆像素的组合。如果那些像素看起来 像个有功能的程序，他们就会认为：「哦，要让这个程序真游动起来应该不会有 多难吧？」这里有一个很大的风险：如果你先做出使用接口的原型，认为这样 就能与客户进行讨论，结果每个人都会认为你几乎都做完了。接下来你花了整整 一年去做「里面」的事，却没人会看到你的工作成果，大家还以为你什么都没做。

重要的推论三：同样是网络公司，比起功能齐全又累积了 3700年资料但用灰色底 色的网站，只有四个网页但外观漂亮的会获到较高的评价。

噢，等一下，网络公司已经完全没有价值了。别在意了。

重要推论四：因为政治因素要求由各技术经理或客户「启动」项目时，可以提供 数种美术设计让他们选择。

改变某些组件的摆设方式，改变外观和字型，移动标志位置，标志也可以变大或 变小。拿些无关紧要的家家酒内容给他们玩，让他们觉得自己很重要。这些他们 就不会严重影响你的时程了。好的室内设计师会定期拿些样本之类的小东西给 客户选，不过从来不会跟客户讨论洗碗机的位置。不管客户想怎样，反正洗碗机 就是放在水槽边，没必要浪费时间去争论，就是得放水槽边，连禮高一^?都免 谈；客户想玩设计就让他去碰些无害的东西，比如流理台面要选意大利花岗石， 还是用墨西哥瓷砖还是挪威木砧板，这种事他改变主意200次都没关系。

重要的推论五：展示时唯一重要的就是画面。一定要让它美得冒泡。

不要认为你能成功地让在何乂去邀康这会有多磨，也不要认为他们会注意功能。 他们才不管呢。他们只想看到一堆漂亮的像素。Steve Jobs懂这个。哦，小朋 友他是真的很懂。苹果的工程师已经学会做出漂亮的画面，看看Dock上那个 1024x1024的华丽新图示（虽然这样会浪费珍贵的桌面空间）。而Linux的桌面帮却 为半透明的xterm而疯狂，画面是很漂亮没错，不过通常用起来很烦。每次当 Gnome或KDE宣布发行新版，我会直接去看看画面抓图，然后说：「噢，他们把行 星由木星改成土星了，酷哦。」不过从来不管他们实际上做了些什么。

记得这篇文章一开始时的执行长吗？他之所以不高兴，是因为团队在刚开始就给 他看很漂亮的投影片，里面都是用外otostopf故的原型(连VB都不是）。而后来都 在实际做底下的东西，所以看起来就像完全没做事。

你要怎么处理这种事呢？当你了解了冰山的秘密后就很容易处理了。要知道在暗 室里用投影机所做的任何展示澈只是像素雨己。如果可以的话应该把未完成的使 用接口画得像未完成。举例来说，在功能完成之前对应的工具栏图示就只用草图。 如果是建立web服务，在功能完成之前就先不要放在首页里。这样大家就会逐渐 看到首页由三个命令扩充到二十个命令。

更重要的是要确定你控制了大家对时程的想法。提供一份Excel格式的详细时程。每星期都送出自我庆祝的电子邮件，谈论本周进度如何由32%进步到35%完成，可以及几勒在十二月廿五日发行。不要让项目是否正常进行的想法影响到真正的事实。 另外不管有多希望老板赢，千万别让他用Callaway的钛制木杆，美国高尔夫协会 已经禁止这种球杆，这真是不公平啊。

二十六、漏洞抽象定律

你每天不可或缺的Internet里有个关键的小魔法，这个魔法就在TCP通讯协议这 个internet的基础协定里。

TCP是一种可靠的数据传输方法。我说可靠是指如果用TCP在网络上传一个讯息， 讯息一定会到，绝不会乱掉或坏掉。

TCP的用途很多，比如抓取网页数据或传电子邮件都是。由于TCP这么可靠，连那 些挪用钱的东非人电邮(译注：指有阵子常见到的骗人信)都能完整无缺的到达， 真是好笑。

相对的有另一种叫IP的不_数据传输方法。ＩP不保证数据会传到，就算到了数 据也可能会乱掉。如果你用IP传送一堆讯息，很可能只有一半的讯息到达，而 且其中还有一些到达的顺序和原先传送时的顺序不同，另外可能有几个讯息的内 容会变掉，可能变成可爱的猩猩宝贝照片，更可能变成一堆看不懂的垃圾，看 起来就像台湾垃圾信的标题一样。

这里就是魔法所在：TCP是架在IP上面的。换句话说，TCP不得不靠一个不可靠游 工為想办法可靠地传送数据。

为了说明这的确是个魔法，想想下面这个本质上相同（虽然有点滑稽），来自真实 世界的情节。

想象你有个方法把演员由百老汇送到好莱坞，基本上就是让人坐上车后开车横越 国家送过去。有些车会出车祸让可怜的演员挂掉。有时候演员在路上喝醉了就 去剃光头或刺纳粹刺青，结果变得太丑而不能在好莱坞工作。另外由于走的路线 不同，演员到达的顺序常会跟出发的顺序不一样。现在想象有个叫好莱坞快递 的新服务，可以把演员送到好莱坞，并且保证演员一定会(a)到达，并保证(b) 顺序不变而且(c)状态完美地到达。神奇之处在于好莱坞快递除了原本的车子以 外，并没有新的运送方法。好莱坞快递的作法是在每个演员抵达时检查演员的状 况，如果状况不佳就打电话请公司把该演员的双胞胎送来。如果演员到达的顺序 不对，好莱坞快递会照正确顺序重新排好。如果51区有架大幽浮在内华达的高速 公路上坠毁阻断了交通，预定走这条路线的演员就会改走亚历桑那州，好莱坞快 递甚至不会把事情告诉加州的导演。导演只会觉得演员来得比平常慢，他们甚 至不会所至幽浮失事的消息。

TCP的魔法大致上就是这样。这种作法常被计算机科学家称为猶康：把复杂许多 的东西隐藏起来的一种简化动作。结果很多计算机程序的设计都是在建立抽象 机制。字符串链接库是什么？它是一种伪装，假装计算机能像处理数字一样轻易 的处理字符串。文件系统又是什么？也是一种伪装，假装硬盘并不是一堆不停 旋转，可以储存位的磁性盘片，而是一个有着层层目录的阶层式系统，可以存放 一个个由一或多个字节字符串构成的档案。

把话题拉回TCP。稍早为了让事情单纯一点，我撒了一个小谎，而且现在有些人

可能会因为这个谎气得头上冒烟。我说过TCP保证你的讯息会到达，其实并不会。 如果你养的蛇把连接计算机的网络线咬断了，就没存任何IP封包可以通过，这时 候TCP当然也不可能让你的讯息抵达。如果你惹毛了公司的系统管理员，他们为 了报复就把你接到已经超过负荷的集线器，因此只有部份的ＩP封包能通过，这时 候TCP是会动，不过一切都会变得很慢。

这就是我称之为猶象树激存漏源的状况。TCP试图提供一个完整的抽象机制，想 隐藏底下不可靠的网络，不过有时候网络会渗漏越过抽象机制，这时就会觉得抽 象其实并不太能真的提供保护。这只是我所谓「抽象渗漏法则」的一个例子而已：

所有重大的抽象机制在某种程序上都是有漏洞的。

抽象会失效。有时候轻微有时候很严重，反正就是有漏洞。事情会因而出错，而 且当你有抽象机制时到处都可能会发生。下面有一些例子。

1. 像扫描一个大的二维数组这么简单的动作，是由水平方向或垂直方 向扫描都会严重影响效率，影响的大小依「木纹」（译注：二维数 组排列的方式)的方向而定，某个方向可能比另一个方向多产生许 多的分页失败，而分页失败是很慢的。虽然写汇编语言的程序员应 该可以假设自己拥有可连续寻址的内存空间，不过虚拟内存表示 这种假设只是种抽象机制而已。当出现分页失败时或是某些内存读 取时漏洞就会出现，处理时间会比其他内存慢几毫微秒。
2. SQL语言希望把数据库查询的程序抽象化，让你只要定义想要的东 西，查询动作的细节就交由数据库去处理。不过在某些状况下，有 些SQL查询比逻辑上相等的查询慢上几千倍。这有个很有名的例子， 在某个SQL服务器用”where a=b and b=c and a=c”来查询，会比用 “where a=b and b=c”快上许多，可是查询的结果其实是一样的。 照道理只要指定规格，并不需要在意程序。可是有时候抽象机制会 失效并导致很差的效率，于是你就得跳出来用查询规划分析器找 出问题，然后想办法加快查询。.
3. NFS或SMB之类的网络链接库，能让你「像」处理本机档案一样地处 理远程机器的档案。有时候连接速度会变得很慢或是断线，这时远 程档案就不再像是在本机上了，而身为程序员的你必须加程序代码 来处理这种状况。「远程档案和本地档案一样」的抽象机制出现漏 洄了。这里有个Unix系统管理员的具体例子。如果你把用户的home 目录放在用NFS挂入的磁盘上(一种抽象机制），而使用者建了一 个.forward档案把他们的电邮全部转寄到其他地方(另一种抽象 机制），如果新邮件进来时NFS服务器停掉了，由于找不到.forward 文件讯息并不会被转寄出去。这个抽象机制的漏洞就真的会把一 些讯息丢掉。
4. C++字符串类别应该能让你假装字符串是个第一级(first-class) 资料。它们尝试把「字符串很难处理」这个事实抽象掉，让它使用 上像整数一样容易。几乎所有C++字符串类别都会多载+运算符，才 能把字符串连接写成s + “bar”。不过你知道吗？不过怎么努力， 世上还是没有C++字符串类别能让你写成”foo” + “bar”，因为C++里的字符串常数一定是char*，绝对不会变成字符串。这个抽象机 制呈现一个程序语言本身不给补的漏洞。（有趣的是，C++随时间演 进的历史，可以描述成尝试用修补字符串抽象机制漏洞的过程。他 们为什么不直接在语言本身加个原生的字符串类别？这实在让我 搞不懂。）
5. 再来就是下雨天时开车没办法开得和平常一样快，虽然车上有挡风 玻璃雨刷有头灯有车

> 顶还有暖气，这些装备应该是让你可以忽略下 雨这个事实（他们把天气抽象化了），不过看吧，你还是得担心天 雨路滑，有时候雨甚至会大到你看不远，所以在只好慢慢地开，因 为基于抽象渗漏法则，天气永远不能完全被抽象化。

抽象渗漏法则会造成问题的原因之一，是因为它说明了抽象机制并不真能照原构 想简化我们的生活。当我想训练某人成为C++程序员时，最好能完全不教char 和指标运算，直接去学*STL*字符串。问题是总有一天他们会写出*"foo"* + *"bar"* 这样的程序然后看到怪事出现，于是我就得停下来教他们有关*char*的事情。他 们也可能会试着呼叫某个需要OUT LPTSTR参数的Windows API函数，于是又得把 char*、指标、Unicode、wchar_t以及TCHAR含入档搞懂，才会知道如何呼叫。而 这些全都是漏洞。

在教COM程序设计时，最好只要教学生如何使用Visual Studio的精灵和各个程序 产生功能。不过万一出了任何问题，他们根本不会知道怎么回事，也不知道如何 除错或回复。我还是得教他们I Unknown和CLSI D还有ProgIDS以及。哦，饶了我吧！

在教ASP.NET程序设计时，最好只要教学生可以在组件上双击，然后就能撰写用 户点击该组件时在服务器执行的程序。不过处理超链接(`<a>`)点击事件的HTML 程序，和某个按钮被按时的处理程序是不一样的，而ASP.NET实际上是把这之间 的差异抽象化了。问题来了，ASP.NET的设计者必须把HTML无法由超链接传送表 格的事实隐藏起来。他们的做法是在超链接的onclick产理加上几行JavaScript 程序。不过这种抽象机制也有漏洞，如果用户关闭JavaScript功能，ASP.NET 的应用程序就不能正常的运作了，万一程序员又不了解ASP.NET抽象掉什么东西， 根本不可能知道出了什么问题。

抽象渗漏法则表示，当某人发明一套神奇的新程序产生工具，可以大幅提升效率 等等，就会听到很多人说：「应该先学会如何手动进行，然后才用这个神奇的 工具来节省时间。」程序产生工具假装抽象掉某些东西，和其他所有抽象机制 一样都有漏洞，而唯一能适当处理漏洞的方法，就是弄懂该抽象原理以及所隐藏 的东西。所以抽象机制虽然替我们节省了工作的时间，不过学习的时间是省不 掉的。

而这一切都似非而是地表示，即使我们拥有愈来愈高阶的程序设计工具，抽象化 也做得愈来愈好，要成为一个纯熟的程序员却是愈来愈难了。

我第一次去微软实习时，写了一个在麦金塔执行的字符串链接库。那是一个很典 型的任务：写一个自己的st「cat函数传回指向新字符串结尾的指针。只要写几行 C就够了。我做的每件事都写在K&R里面(一本讲C程序语言的薄书）。

今天为了要做CityDesk，我必须会Visual Basic、COM、ATL、C++、InnoSetup、Internet Explorer内部机制、正规表不式、DOM、HTML、CSS以及XML。一大堆比古老的K&R更高阶的工具，可是我还是得会K&R讲的东西，否则我就完了。

我们十年前可能想象过，现在会有某些全新的程序设计典范让程序设计更容易。 事实上这些年间所建立的抽象机制，游磁让我们能处理更高复杂度的软件开发 (如GUI程序设计和网络程序设计），这是十或十五年前无法处理的。这些伟大的 工具（比如00型式的程序语言）虽然

能让我们用飞快的速度完成许多工作，不过 总会有一天我们得去追查因抽象渗漏而产生的问题，到时候就得查上两星期了。 另外虽然你得雇一个以写VB程序为主的程序员，不过单纯的VB程序员是不够的， 因为当VB的抽象机制渗漏时他们就完全卡住了。

抽象渗漏法则正在拖垮我们。

二十七、程序设计界的LordPalmerston

曾经一度如果读过Peter Norton的一本书，就能完全了解在IBM-PC上写程序所需 的全部知识。在过去20年间，全世界的程序员很努力地在IBM-PC上建立层层的抽 象机制，让程序更容易撰写而且功能更强大。

不过抽象渗漏法则表示，即使他们建立了这些理应让程序更易设计的抽象机制， 做个伟大的程序员所需的各种知识还是一直在增加。

要真正地精通某个程序设计领域需要好几年的工夫。当然也有很多出色的青少年 学了一星期Delphi，再学一星期Python然后再学一个星期的Perl，就自认已经精 通了。不过他们根本不知道自己差得有多远。

我打从ASP和VBScript刚出来就在用了。VBScr i pt是世界上最微不足道的程序语 言，而ASP程序设计大概只要上五堂课会了，而且其中只有两堂是会经常用到的。 可是直到现在，我才感觉自已知道架构一个ASP/VBScript应用程序的最佳作法， 我才终于觉得自己知道数据库存取程序最适当的位置，用ADO取得recordset的 最佳作法，分离HTML和程序代码的最佳方式等等。另外我也终于会用正规表示式， 不再每次都重写一次字串操作函数。另外我上星期才学会如何取得不在内存中 的COM对象，这样就就可以不需整个web服务器重开就能重新编译。

Fog Creek太小了养不起专家，所以当我要为FogBUGZ(我们用ASP/VBScript制作 的产品）写一个真正好的安装程序时，我得拿出数年C++/MFC和Windows API的经 验，再用我还不错的Corel PhotoPaint技巧做出放在精灵角落的图片。另外为了 让FogBUGZ能无误地使用Unicode，我还得用C++和ATL写一个小小的ActiveX组件， 这里头有着数年的C++和COM经验，还有当初实作CityDesk时学了一周左右的字符 编码知识。

所以当我们遇到某个只有NT 4.0才发生的怪问题时，我只花了三分钟就解决了， 因为我知道如何使用VMWa「e，而且我有一套用VMWa「e灌的干净NT4.0机器， 另外 我也知道如何用Visual C++做远程除错，还知道可以由EAX缓存器知道函数的回 传值。完全没接触过这些东西的人要抓同一个问题，恐怕得花一个小时或更多的 时间，不过我已经知道了很多很多「东西」，那些基本上是从1982年拿到我第 一台IBM-PC和那本Norton书时就开始学的东西。

有漏洞的抽象表示我们面对一个直线上升的学习曲线：你可以用一星期学到每天 工作所需知识的90%。不过其他10%可能得要好几年才能补齐。有些人会说：「不 管你要我做什么，我都可以拿本书来就学会了。」真正有经验的程序员超越这种 人的地方就在这里。如果你正在建立一个团队，当然可以找一堆经验较少的程 序员用抽象工具制作出一大堆程序代码，不过如果少了经验老到的人去做真正困 难的事情，这个团队是做不起来的。

程序设计有很多不同的世界，每个世界都需要大量的知识才能真正的精通。以下 是我个人最熟的三个领域：

1. MFC/C++/Windows
2. VBScript/ASP
3. Visual Basic

基本上这全部都可以称作Windows程序设计。没错，我也写过Unix或Java的程序， 不过很少就是了。我对Windows程序设计的专精不光是了解基本技术，也知道整 个支持的整体结构。我敢宣称对Windows程序设计很在行，是因为我也会COM、ATL、 C++、80x86 汇编语言、Windows API、IDispatch (OLE Automation)、HTML、DOM、 Internet Exp lore「对象模式，Windows NT以及 Windows 95 内部机制，LAN Manager 和NT网络以相关安全机制（ACE、ACL等等）、SQL及SQL Server、Jet和Access、 JavaScript、XML，还有一些其他有关直角三角形斜边的有趣事实。如果不能在 VB里用StrConv函数达成我的目的，我会为了用C++和ATL呼叫MLang函数而去写一 个COM控制组件，眼睛连眨都不会眨一下。我花了很多年才达到这种境界。

还有很多其他程序设计的世界。有开发BEA Web log i c的世界，里头的人要懂J2EE、 Oracle还有各种我列不出来跟Java相关的东西。另外也有硬派的麦金塔开发者， 他们懂的是CodeWarrior、MPW、由System 6到OS/X各版本的Toolbox程序设计、 Cocoa、Carbon、甚至还有现在已经没用的OpenDoc等过时的好东西。

不过只有极少数人能清楚一个以上的世界，因为要学的实在太多，除非在这些世 界工作过几年，否则是不会真的全部都懂。

不过你却必须学会。

面试时因为没有Win32或是J2EE或Mac程序设计经验而被拒绝，大家都会觉得很生 气。或者某些笨蛋面试官其实根本不知道MSMQ，却打电话问面试者是否具有「五 年的MSMQ经验」，也会让人觉得很不爽。

如果你刚写Windows程序不久，你可能认为Win32只不过是个链接库，跟其他链接 库差不多，要用到时再去查书学着怎么呼叫就好了。你可能会认为基本的程序 设计（比如你的C++专业技能）占九成，所有API合起来算是只占一成的小事，几星 期就可以恶补回来。对这些人我只能小声地建议：时代已经变了。现在的比例 是反过来的。

已经很少有人需要去做把字节搬来搬去的低阶C算法了。现在我们大多数人都把 全部时间花在呼叫API而不是搬移字节。一个没有API经验的C++程序高手，对用 API写程序的日常工作内容其实只知道大约一成而已。当经济状况好的时候这并 不打紧。你还是会得到工作，而雇主会花钱让你熟悉平台。不过当经济不景气时， 每个工作都有600个人申请，雇主有本钱选择已经精通该平台的程序员。比如说 能列出四种Visual Basic中FTP档案的方法，还能指出每种方法优缺点的人。

这所有的程序设计世界的表相都很多，因此常常引发毫无意义的论战，争执哪个 世界比较好。某位匿名者在我的讨论区写了以下这段自以为是的意见：

「再说另一个让我乐意留在『自由世界』的理由，言辞的自由（几乎），以及免于屈从安装程序和 registry这种东西(还有很多，我只是举两个例子)的自由。」

我认为这个人真正要说的是，在Linux世界里是不写安装程序的。嗯，我讨厌让 你失望，不过你的世界也有些一样麻烦的东西：imake、make、config檔等玩意。 另外当你完成应用程序之后，要发行时还得附上20KB的INSTALL档案，里头全是 些拿古怪装有趣的指示，比如「你会需要zlib」（那是什么东西？）或是「这会 要花一些时间，去拿一些runt吧。」（我猜runt大概是某种糖果吧）。至于registry 登录档，虽然你并没有用一个有组织的大结构(hive)存各个名称/值的组合，不 过你有上千种不同的文件格式，每种应用程序都各有一种，而且到处都有.「c和 foo.conf档案。另外想在emacs里更改设定还得学写lisp程序，而每个shell都 要你学它独有的shell脚本方言才能更改设定，还有其他其他其他。

只认识一个世界的人是很讨人厌的。他们每次听到其他世界的复杂状况时，就会 觉得自己的世界没那么复杂。不过事实上是一样的。只是你已经很精通，所以 视而不见。这些世界实在太大太复杂无法相互比较。帕麦尔斯顿勋爵(Lord Palmerston)说过：「什列斯威-好斯敦问题（Schleswig-Holstein Question)问 题实在是太复杂了，整个欧洲也只有三个人了解。第一位是已经逝世的Prince Albert。第二位是某个疯了的德国教授。第三位就是我，不过我已经完全忘掉了。」 软件的世界也是如此巨大复杂而多面相，以致当我看到某些其方面很聪明的人 在网志写些像「微软不懂操作系统」之类的空话时，不禁要坦白的说他们看起 来很蠢。Windows是由数百位程序员在十到廿年间所创造，具有数百个功能的几 百万行程序。想象尝试要概述这种庞然大物，却连开始了解其中一大部份就没 有。我并不是在为微软辩护，只是觉得这种源自极度无知的巨大模糊概括，是现 在网络上最浪费时间的东西。

常来的读者应该已经注意到我在思考要如何发展跨Linux、Macintosh和Windows 平台的应用程序，又不必为Linux和Macintosh版本投入不成比例的成本。这时候 你需要某种跨平台链接库。

Java有这个野心，不过Sun并没有真的把GUI弄通，不能做出真正感觉自然的应用 程序。这就像星舰迷航记里的太空异形透过望远镜观察地球一样，他们知道人类 食物的外观，却不了解它的味道。Java程序会在正确的地方画出菜单，可是键 盘操作和Windows程序完全不一样，而且Java的分页对话盒看起来有点恐怖。另 外不管怎么试都不能让菜单栏和Excel的一模一样。为什么会这样呢？因为Java 在抽象机制失败时，并没有提供很好的方法去将就使用平台原始的机制。当你用 AWT写程序时并不能取得窗口的HWND，不能呼叫微软的API，当然也不能把 WM_PAINT拦截起来自己画。而Sun已经表示得很明白了，如果你这样就就不够纯粹。你被污染了所以活该去死。

试图用Java建立GUI的努力经过多次众所周知的失败之后（比如Corel的Java Office suite和Netscape的Javagator)，不少人学会要和这个世界保持距尚。 Eclipse运用平台原本的机制，从最底层开始建立了他们自己的窗口链接库，这 样才能写出使用观感符合原平台感觉的Java程序。

Mozilla的工程师决定用他们自己的发明XUL来处理跨平台问题。到目前为止令我 印象深刻。Mozilla终于做到用起来感觉对劲的境界。连我最喜欢的怪用法「用 Alt+SpaceN把窗口缩到最小」都可以在Mozilla里面使用；虽然花了很久不过他 们还是做到了。

成立Lotus并创造123的Mitch Kapor决定下一个计划要做一个叫wxWindows和 wxPython的产品，目标也是跨平台支持。

哪一个比较好呢？是XUL?是Eclipse的SWT?还是wxWindows呢？我不知道。这些 都是很庞大的世界，大到我根本无法实际去评估并找出答案。光是读完教学并 不够。你必须自己去辛苦用个一两年，才能真正知道它是的确够好，还是怎么试 都无法让做出感觉对劲的使用接口。不幸的是，大多数项目都必须在写第一行 程序之前决定要用哪个世界，而这正是信息最少的时候。在前一个工作我们得面 对某些很烂的架构，因为最早期的程序员利用该项目同时学写C++和Windows程 序设计。有些最初期的程序代码是完全不懂事件驱动程序设计下写出来的。核心 的字符串类别(我们当然有自己的字符串类别)可以写进教科书，用以示范C++

类别在设计上会犯的各种错误。最后终于把很多旧程序清理并重整干净，不过还 是让我们头痛了好一阵子。

所以现在我会建议：至少要有一个对所用的语言、类别、API以及平台有数年以 上经验的设计者，否则还是不要启动项目吧。如果你可以选择平台，就用你的 团体最熟悉的吧，即使这个平台并不是最符合趋势或看起来最有生产力也没关系。 另外在设计抽象机制或程序设计工具时，多做些努力让它不会漏吧。

二十八、评测

「谢谢你打电话来Amazon.com，有什么事可以为您服务吗？」然后…嘟一声！电 话就被挂掉了…这实在让人生气。等了十分钟才转接给一个真正的人听，结果电 话却离奇的马上断线。 不过这真是是离奇吗？依据Mike Daisey的说法，Amazon是依据每小时接听电话 数量来考核客户服务代表。所以提升考绩最佳的方法就是挂掉客户的电话，这样 才能增加每小时能接听的电话数目。

你说这只是个案吧？

当Jeff Weitzen入主Gateway时，他订了 一个节约客户服务电话费用的新政策。 「客服人员如果和一个客户谈超过13分钟，就拿不到当月的奖金，」Katrina Brooke「在Business 2.0(2001四月）写道：「于是工作人员就无所不用其极要让 客户挂断电话：假装电话线坏掉或断线了，或者(让公司损失更多地)给客户换新 零件或新电脑。无怪曾为业界顶尖的Gateway客户服务满意会掉到一般水平以 下。」

似乎每次想要测量知识工作者的绩效时，情况就会急速走样，然后你就会得到 Robert D. Austin所说的测量机能障碍现象。他的书组织中绩效的测量与管理对 这个主题有出色而彻底的调查。经理人喜欢施行测量系统，而且喜欢把它和奖惩 方案绑在一起。不过只要没有百分之百的监督，工作人员就有诱因来个「下有对 策」，心里只想着那个测量系统，完全不顾工作的实际价值或质量。

软件组织通常会奖励(a)写大量程序以及(b)修正大量问题的程序员。想在这种组 织中出头，最好的方法就是写一大堆错误百出的程序之后再全部修好，而不是 多花些时间在一开始就把东西做好。如果想惩罚产生问题的程序员来避免这个问题，就为他们创造了一个堕落的动机，让他们努力隐藏问题，或者写了程序却 不告诉测试人员，好让测到的问题少一点。你是不可能赢的。

财星前500大公司执行长的报酬通常都是底薪加股票选择权。股票选择权通常价 值数千万或数亿美元，相较之下底薪几乎无关痛痒。结果执行长就会花招百出 炒高股价，即使会毁掉公司或让公司破产都在所不惜(就像这个月一直上头条的 新闻一样）。即使股价只是暂时上升也照做不误，因为他们会在高点卖掉。奖惩 委员会的反应很慢，不过他们最新的好点子是要求执行长要一直持有股票，直到 离开公司才能脱手。了不起啊，现在等于要他们短线炒高股价然后赶快走。同 样的，你也是不可能赢的。

就算我乱讲好了，你也可以去看看Austin的书，就会了解为什么无法彻底监视工 作人员时(这几乎是常态），这种测量机能障碍现象是绝对无法避免的。

我一直都声称奖励并不是那么好的主意，即使能够度量谁有或没有把工作做好也 一样。而Austin指出连测量绩效本身都有困难让加强我的想法，所以激励奖赏就 更没有作用了。

第三部分 Joel对常态问题的遐想

二十九、RickChapman解读愚昧

(这是Rick Chapman的新书寻找，愚蠢的序言。）

在每个我知道的高科技公司中，都在进行一场计算机狂和西装客间的战争。

这本好书里充满了软件营销魔法师兼终极西装客Rick Chapman的宣传，在你开始 阅读前请容我花一点时间提出计算机狂的想法。

陪我玩一分钟，好吗？

请想象最刻板的印象，苍白，喝Jolt可乐(译注：美国黑客最爱的饮料，咖啡因 很高），吃中国食物，玩电视游乐器，看slashdot，住在Linux命令行里的呆子。 由于这只是一个虚构的刻板形象，你应该可以自由想象成小矮子或胖子，不过两 种都不属于会和高中同学踢足球，感恩节会去看妈妈的类型。另外由于只是个形 象，我也就不必找些麻烦的借口让它变成餚了。

我们这个虚构的程序员会想：「微软做的产品不好，可是他们营销一流，所以大 家都买他们的东西。」

问问他对自己公司营销人员的看法：「他们真的很蠢。昨天我在休息室跟某个笨 业务菜鸟大吵了一顿，吵了十分钟才搞清楚她椒本不懂802.11a和802.116的差别。 哼！」

那些营销人员都在干什么呀？年轻的计算机狂。「我不知道..他们陪客户打高尔 夫球或是做别的事吧，不然就是在写那些白痴规格书，害我要一直帮他们订正。 如果我能作主，我一定会把他们全部开除。」 有位叫Jeffrey Tarte「的好人一直在出版一份叫Soft-letter 100的刊物，列出 每年排名前一百的个人计算机软件厂商。以下是1984年前十大的名单[1]:

看到了吧，微软排第二，不过收益跟其他几家公司差不多。

现在让我们看看2001年的同一份名单。

排名	公司	年收益
1	Micropro International	$60,000,000
2	Microsoft Corp.	$55,000,000
3	Lotus	53,000,000
4	Digital	Research $45,000,000
5	VisiCorp	43,000,000
6	Ashton-Tate	$35,000,000
7	Peachtree	$21,700,000
8	MicroFocus	$15,000,000
9	Software Publishing	$14,000,000
10	Broderbund	$13,000,000

哇！注意微软以外的每一家公司都不在前十大了。另也也请注意微软比次大的公 司大/淨多，这并不好笑。Adobe光是拿到微软苏打汽水预算，就可以让自己的 年收益倍增了。

个人计算机软件市场就等于是微软本身，微软占了前100大公司总收益的69%。 这就是我们要在这里谈的东西。

原因真如我们虚构的计算机狂所言是营销做得太好吗？还是违法垄断的成果？(这回避了下面的问题：微软如何取淳这种独占地位？你不可能两个问题都对 吧。)

Rick Chapman有个更简单的答案：微软是名单上唯一没有犯致命蠢错误的公司。 不知道是因为脑力优异或只是狗屁运，微软犯过最大的错误是那个在跳舞的回形 针。而这个错误真游很糟吗？我们都在嘲笑并把它关掉，然后还是时时刻刻都 在用Word、Excel、Outlook和Internet Explore「。不过对于其他曾在市场上领 先却又消声匿迹的公司，你都可以指出一两个足以让船撞上冰山的巨大失策。 Mic「op「o败在重写打印机架构而不去更新旗舰产品WordStar。Lotus浪费了一年 半去削足适履，让123能在640 KB的机制上执行，在他们完成之前Excel早已推出 而640 KB的机器也已成为往事。Digital Research疯狂地对CP/M-86出价过尚， 错失成为PC操作系统业界标准的机会。VisiCorp则是把自己告倒了。Ashton-Tate 从不错失把dBase开发者赶走的机会，持续毒害对平台厂商成功如此重要的脆弱 生态。

我是个程序员（当然啦），所以我倾向把这种愚蠢的错误算在营销人员的帐上。绝 大多数的营销人员都和非技术性的商业界人士一样，无法了解基本的技术事实。 当百事可乐推手John Scul ley发展Apple Newton时，他并不知道当时国内每个计 算机科系学生都知道的事：手写识别是不切费游。而同一时期比尔盖茨却把程 序员拉来开会，恳求他们写一个能在所有产品中重用的rich text编辑控制组 件..如果让Jim Manzi (让MBA入主Lotus的西装客)也参与那场会议，他可能会 满脸迷惑的说「什么是rich text编辑控制组件？」他永远不会在技术上取得领先地位，因为他没有搞懂(grok)技术grok;事实上光是在句子里用grok这个字可 能就已经让他完全出局了。

如果你问我并让我偏私的回答，我会说除非有个程序员掌舵，否则没存表■洋公司 会成功游。到目前为止的证据都可以支持我。不过这些要命的错误中有许多也是 程序员搞出来的。Netscape重写浏览器而不去改善原有的程序代码库，这个不朽 的神经病决定花了他们好几个年的Internet时间，在这期间他们的占有率由大约 九成掉到4%左右，而这是涯^序屑游点子。当然啦，该公司里面非技术又无经验的 管理阶层完全不知道这为//■么是个烂点子。还是有很多程序员为Netscape彻底重 写辯解。「旧的程序真的很烂耶，约耳！」是，嗯哼，这种程序员应该因热爱干 净的程序而被赞赏，不过绝对要禁止他们接近任何商业决策100呎的范围内，因 为对他们来说，干净的程序代码显然比让软件出货重要多了。

所以我会对Rick稍微让步地说，如果你想要在软件业成功，必须有一个完全了解 并热爱程序设计的管理团队。不过这个团体也必须了解并热爱商业。要找到对 两边都很有天赋的领导者并不容易，不过只有这样才能避免这些被Rick亲切地列 在书中的错误。所以读吧，笑一笑。如果你的公司是某位猪头在经营，就把履 历整理一下开始在Redmond(译注：微软公司所在)找房子吧。

三十、在这个国家狗是干什么的？我们有多么天真?

我们假设Bezos(译注：Amazon的执行长)只是把收益重新投入，所以才没有从谷底爬起来。

去年大概这个时候，第一家大网络公司倒闭的消息开始见报。Boo.com和Toysmart.com都关门了。快速变大的想法不再适用。Dockers里的五百名年轻人发现光是照抄Jeff Bezos并不 能算是商业计划书。

Fog Creek过去几星期出奇的安静。我们正在做CityDesk的收尾动作。我想告诉你CityDesk的一切，不过得再得一阵子。现在要谈关于狗食的事。 狗食?

上个月Sara Corbett写了关于迷惑的男孩的事。一群8到18岁与家人分离的苏丹难民，被迫 跋涉千里由苏丹到衣索比亚再回苏丹再到肯亚。有一半的人在旅途中因饥渴或短吻鳄而死去。有些人在冬季中期获救后被送到北达科塔州Fargo等地。其中一个难民由机场乘车到新家时 问道：「丛林里有狮子吗？」

彼得触碰我的肩膀。他拿着一罐普瑞纳狗食。「对不起，Sara，不过你可以跟我说这是 什么吗？」他后面的狗食由地板一直堆到屋顶。我回答：「呃，这是我们的狗吃的食物。」 一面担心八年来都吃粥维生的人听起来会怎么想。彼得说「哦，我知道了。」他在罐头放为 柜上，似乎对我的答案觉得满意。他推着购物车走了几步，然后转过来困惑地看着我说：「告诉我，这个国家的狗做什么工作？」[纽约时报杂志，四月一日，2001]

狗。没错，彼得。Fargo的食物充足，连狗也一样。

真是令人沮丧的一年。

噢，刚开始时多么好玩，我们全都投入B2B和B2C还有P2P准备大干一场，像一个搬到效区的快乐家庭星期天去麦当劳玩。不过等一下，这并不是好玩的部份，真正好玩的是看着最烂的商业计划书失败，然后股价由316掉到3/16。接招吧，新经济的大嘴巴们！哈，惨得真好。哈，真爽，Wired杂志又再次证明，只要把某样东西放在该杂志的封面，这个东西就会在短 短几个月内被证明是又笨又错的。

哦？抱歉，你买了联机指数(Wired Index)吗？

至于新经济这东西，Wired真的把它毁了。报导了嗅觉技术(smeM-o-「ama)和注定失败的游戏公司那么多年，又说过PointCast会取代web(等一下，PointCast不是已经在1997年三月取代web 了吗)这种话，他们应该早己知道他们的封面对任何技术或公司甚至一昔成名之物都是个死亡之吻。不过反正他们就是不信邪，不只把新经济放在封面上，还把该死的整整一期都拿去写新经济，所以把NASDAQ搞到像学飞的羊一样直线下跌。 不过我们以他人的不幸为

乐，也只能高兴到现在。如今情况愈来愈沮丧，我知道经济并不是 真的肃条，不过我本身却还是很沮丧。并不是因为有这么多愚蠢的新公司倒掉，只因为整个 思潮很沈闷。所以现在我必须弃麦当劳而改吃狗食。

这就是我们正在做的事情，因为日子常还是要过。每个人都垂头丧气地到处乱走，抱怨为了SockPuppet.com的股票选择权投入了多少小时，如何牺牲健康和爱情生活，即使如此生活还是照常要过。产品开发循环当然也得继续，而在Fog Creek的我们也走到产品开发循环中要吃自己狗食的那个过程。所以短期内我们会是Dog Creek Software。

在计算机业界中，吃自己的狗食是个怪名称，表示真正使用自己产品的动作。我直到一个月前都忘了它的效果有多么好，我拿了某一版的CityDesk回家(想着差三星期就要推出了），然后尝试用它建立一个网站。

哎呀！结果发现有几个问题，根本不可能开始测，得先把问题修好才能继续试。我们所做的测试(小心地点选每个选单，看看是否会动)并没有发现这些让产品无法达成预期功能的严重问题。而像个顾客一样地使用产品却在一分钟内就找到了。

还不只这些呢。我继续静静地想建个简单的网站，什么功能都还没执行，光在星期天下午就找到45个问题。而我是个懒人，不会在这件事上用超过两小时。我只用了产品最基本的功能， 其他东西连试都还没试。

我星期一早上回去工作时就把整个团队都找去厨房。我告诉他们这45个问题(公正地说，里 面很多都不是真正的问题，只是没有原本设计那么方便的小缺失）。然后我建议大家要用CityDesk至少建立一个够份量的网站，好把问题都熏出来。这就是吃自己的狗食的意思。

以下是找到的某个问题。

我预期很多人会想用复制HTML码并贴上的方式，把现有的网页汇入CityDesk。这个动作是可以正常作用。不过当我想汇入某个纽约时报的网页内容时，我得花一整天耐心地编辑HTML内容，找出所有IMG连结(连到外部的图片)并由网络下载图片，再把图片汇入到CityDesk， 然后把IMG链接改成指向内部图片。虽然很难令人相信，不过那个网站上某篇文章竟然有65 个IMG连结，分别指向35张不同的图片。有的是只有一个像素的分隔图片，小到很难用浏览 器下载。另外CityDesk有个好玩的强制规定，会把汇入的图片用神圣的数字重新命名，而且 完全没办法知道会用什么数字，总而言之就是我得花一整天才能把一个网页汇入CityDesk。

这开始有点令人沮丧，所以我就离开一阵子去除草了。（我不知道等草都除完之后要用什么来排解压力。感谢上帝我们还请不起造景服务。）这时候我突然想到：嘿，我是个程序员耶！ 汇入一个网页加上调整图片用的这些时间，足够我写个全部自动完成的子程序了！事实上， 去写个子程序用的时间可能还比较少。现在汇入一个网页不用一整天，只要半分钟不到就好 了，而且基本上完全不会做错。 哇！ 这说明了为什么要吃自己的狗食。

当Michael自己开始汇入某些网站时，他发现大约十个我不小心引起的问题。举例来说，我 们发现网站用的图片名称如果太复杂，在汇入时会无法转成档名，比如问号在URL里是合 法的，可是不能用在档名里。

有时候当你下载了某个软件，却发现它烂得无法置信，或是费尽工夫才能完成该软件应该很简单就能做好的事。这时候原因很可能就是开发人员并没有用过该软件。

关于没有吃自己的狗食而出事，我还有个更好玩的例子。猜猜看Juno在线服务内部用什么电子邮件产品？[告诉刚来这里的人，我在Juno客户端程序团队工作过几年。]

嗯，你猜Juno吗？因为那是，呃，我们的产品吗？

错。有些人(包括老板)在家里用Juno,不过我们其他175人都用微软Outlook。

至于不用的原因？ Juno客户端程序并不是什么伟大的电邮软件;两年间我们唯一做的事就是用更好的方法显示广告。我们大多数人都想过，如果我们必须用这个产品，我们就得把它做好一点才能让自己好过一点。老板非常坚持在使用过程中要显示六个广告，不过等他回家看过那六个突现式的广告之后，说的话就不一样了 ：「知道吗？两个广告大概就够了。」

AOL的会员人数能急速上升，部份原因是因为它提供比Juno更好的使用者体验，而我们因为没有吃自己的狗食，所以搞不清楚这回事。而我们不吃自己的狗食是因为它很难吃，可是管理阶层实在太失职，根本不准我们修正问题或让它至少可以入口。

总之CityDesk开始感觉好了许多。我们把那些问题全部修好，另外又找到些新问题然后 也全解决掉。我们加了一些之前忘记但后来发现很重要的功能。而且我们愈来愈逼近出货了！ 万岁！谢谢上帝我们不必和37家有着2500万创投资金的公司竞争。这些公司竞争的方式是到 处免费送出产品，好在你前额刺上大片的广告刺青。在这个后新经济年代，大家都想找出不 需要收钱的方法。如果你够聪明的话，后新经济并没有什么不对。不过这数不尽有关于「网 络昏迷(dot-coma)」的新闻，只显示商业刊物编辑的创意不足。fuckedcompany.com抱歉了，玩一个月还算有趣，搞到现在就变可怜了。我们会聆听客户的声音并吃我们自己的狗食，专注改善我们的产品并会坚守本业。而不是在到处飞来飞去募集更多创投资金。

三十一、作为哼哈二将，只管去做事

这个站应该是关于软件管理的，不过有时候你并没有那个权力去改变组织的行政 命令。很显然的，如果你只是图腾柱底层的一个普通程序员，绝对不可能命令大 家开始建立时程或问题追踪数据库。事实上当你成为经理之后，可能会发现管理 开发人员很像养猫，不过没那么有趣。光会说「照这样做」并不真的能照这样做。

在约耳测试分数低的组织里工作可能会让人沮丧。不管你程序写得多好，同事 还是会写出让你羞与为伍的烂程序。另外管理阶层在决定产品时也会作出烂决策， 所以你只好被逼浪费自己的才能，去为针对儿童的AS/400版退休计划游戏除错 (译注：谁家小孩会玩退休计划游戏，更别提能用到AS/400了）。

我建议你大可离开。不过万一你基于某些原因必须继续待着，比如股票选择权还 没有到手，或是住的地方太小没有更好的工作，又或许老板抓了某个你爱的人作 为人质。不管如何，在一个烂团队里讨生活实在叫人抓狂。不过还是有些策略能 从底层改善你的团队，而我要与你分享其中几项。

策略一：去做就是了

有很多事一个人就可以改善项目。没有每日编译服务器吗？架一个。在你自己的 机器上设一个定时的工作，在每天晚上编译并把用电邮发送结果。编译的步骤太 多吗？写个makefile吧。没有人做可使用度测试吗？用几张纸或VB原型自己去找 收发室的人来做吧。

策略二：利用病毒性营销的威力

约耳测试中很多策略都可以在不合作的团队中一个人施行，其中有几项做得好的 话还能推广到整个团队。

举例来说，假设团队中没人愿意用错误追踪数据库。别因此觉得困扰，自己去用 就是了。把你自己程序里找到的问题输进数据库。如果你找到确实是其他人该修 的问题，就用错误数据库把问题指派给他。如果你用了好的问题追踪软件，这时 候就会用电子邮件通知他。没有也不打紧，如果他们没有修正问题，你还是可以 一直寄电子邮件提醒。最后他们总会发现问题追踪的价值，然后像自愿般地开 始使用这个系统。如果QA团队拒绝把问题输入问题追踪数据库，你只要拒绝其他 方式的错误回报就好了。等你重复告诉大家约三千次：「请听我说，我很想修 好这个问题，不过我会忘记，能不能请你把问题输到系统里？」他们就会开始使 用数据库。

团队里没有人要用源码版本管理吗？建立你自己的CVS储存库，必要时可以放在 你自己的硬盘上。即使没有人合作，你还是可以把其他人的程序自己放进系统。 然后当他们发生某些可以用版本管理解决的问题(通常是误把 `rm *?` 打成 `rm *` 时，就会来找你帮助。最后大家都会知道他们自己也可以由版本管理取出程序代 码。

策略三：建立一个卓越圈（**Pocket of Excellence)**

团队不排时程？不写规格？自己来写吧。花一两天针对你的工作写份小规格和时 程，不会有人因而抱怨的。

找些更好的人进团队。想办法参与雇用面试过程并招募好的人才加入团队。

找出那些愿意又有能力改善的人，想办法让他们支持你。即使是再烂的团队还是 很可能有聪明人在，只是缺乏写出好程序的经验。帮助他们脱离困境，安排他 们学习。看他们登入的程序，如果他们做了什么蠢事，不要写电邮高傲地指出他 们的程序哪里不对。这样只会激怒并让他们更加护短。应该故作无知地回报该 错误会导致的问题，让他们去找出原因。等他们自己找到问题，印象就会深刻多 了。

策略四：让笨蛋无害

即使最好的团队都会有一两个笨蛋。团队里有烂程序员让人头痛的地方，就是他 们的烂程序会搅乱你的好程序，另外就是好程序员得花时间帮坏程序员擦屁股。

身为底层工作人员，你的目标是损害最小化，也就是所谓的牵制策略。有时候这 些天才会花两星期写出一点点程序，而且写出来的东西烂到不可思议，完全不 可能用。这时候你会很想花15分钟把这段程序从头再写过。请忍住这种诱惑，因 为这是个把这些白痴拖住几个月的大好机会。只要一直报告那个程序的问题， 他们就只好一直在这段程序卡乂/刀个万，直到你找不到其他问题为止。而这几个 月他们就不会在其他地方造成伤害了。

策略五：远离干扰中断

所有好的工作环境都一样(个人办公室，安静的工作状况，优越的工具，很少干 扰和更少的大型会议）。而所有不好的工作环境都各有不好的原因。

不幸的是，几乎在任何公司里都不太可能改变工作环境。长期租赁表示可能连执 行长都无计可施。这也说明了为何只有少数的软件开发人员拥有自己的办公室。 对公司来说这会造成两种伤害。首先是更难以请到顶尖的开发人员，这些人偏好 能提供更佳环境(当其他条件相等时）的公司。其次是干扰的程度会让开发人员 的生产力急速下降，开发人员会发现自己根本无法进入状况并专心工作。

想办法离开那个环境。拿台笔记本电脑到公司的餐厅去，里面有很多大多时候都 空着的桌子(而且没人找得到你）。可以把某间会议室预约一整天然后在里面写 程序，然后用工作成果来证明，独自在一个房间作业能完成的工作会多上许多。 等下次出现紧急状况，经理要求在明天之前完成工作时，你就知道要说什么了。 他们会找间办公室给你在那天用，然后不久他们就会开始觉得应该整年都保有这种生产力。

上下班都晚一点。其他人下班之后的时间生产力最高。如果所在的开发团队习惯 晚到，你可以早上九点就来，在其他人上班干扰之后专心做事，这段时间的成果 会比后面的时间加起来都多。 不要开着你的电邮和实时通讯。需要的话可以每小时去收一次信，不过不要一直 开着。

策略六：提升自己的价值

如果你并不是真的很有贡献，这些策略通通都没用。如果你写不出好的程序，那 么你「应该」正在写程序的时候，其实只是在问题数据库里浪费时间还惹人厌。 如果你被挂上做不出东西却只关心流程的恶名，大概就没什么职业生涯可言了。

曾经一度当我在某家新公司开始当个底层程序员时，发现公司约耳测试太概只有 两分，于是我决定要改善这种状况。不过我也知道良好的第一印象非常重要，所 以安排自己在每天前七个小时都照大家的期望单纯只写程序。频繁地把程序登 入版本管理，是让开发团队认为你很好的最佳方法。不过我会在每天下午回家前 保留一小时改善流程，利用那段时间修正让产品难以除错的问题。我架了一台每 日编译服务器和一台问题追踪数据库，还把所有长久阻扰开发的烦恼都解决掉。 我针对当天要做的事撰写规格，也写了文件逐步解释如何从头建立一台开发用的 环境。内部有个重要的程序语言没有任何文件，我也把它完整的文件做出来了。 整个流程慢慢地变得愈来愈好。虽然除了我(还我带的小组)之外没有写规格或时 程，不过除此之外约耳测试已经提高到十分左右。

即使你没有当家作主，还是可以让事情变得更好，不过你必须像西泽的妻子一样 无可质疑之处，否则就会一边进行一边树敌。

三十二、两个故事

我要说两个故事，都是我过去工作时发生的。我认为这两个故事足以清楚阐明，管理良 好的科技公司与一团乱的科技公司间有何差别。最终的差异在于相信员工并让他们把事情做 好，还是把他们视为作业员对待，必须不停监视控制以免偏离主题并造成破坏。

我首次工作是在微软，第一份任务是要想出一套新的Excel宏语言策略。很快的我就写 出第一份"Excel Basic"规格初稿（后来发展成Visual Basic for Applications,不过那是 后话了）。不知怎么的，微软里有个叫「应用架构组」的神秘小组听到这份规格。由于他们 认为自己就是负责集语言策略之类的东西，当然得关心这份文件。于是就要求看看我的规格。

我到处问应用架构组是什么来头，不过似乎没有人认为他们是玩真的。原来这个组只有 四个人，是最近才请来的博士(这就微软而言很罕见)。我把我的规格印了一份给他们，还去跟他们开会看看能否听到什么有趣的东西。

其中一个说：「哇啦哇啦！」另一个又说：「哇啦哇啦，哇啦哇啦哇啦！」我不认为他 们讲得出什么有意思的东西。他们非常迷恋子类别继承（subclassing)，似乎认为在Excel 写宏的人会用到大量的子类别继承。无论如何，最后其中一个人说：「不错，这些全都很有 意思。再来还要做什么？要找谁批准你的规格呢？」

我笑了。虽然我在微软只待了几个月，也知道没有什么「某人批准我的规格」这种事。 惨啊，没有人有时间读我的规格，更别提批准了。程序员每天都在催我多写出几页，让他们 能多写点程序。我的主管(还有他的）已经讲得很清楚，没有其他人懂宏或者有时间去做这件 事情，所以不管我做出什么东西，最好是正确无误的。而在微软这个怪研究小组工作的博士 却假设事情会更正式一点。

我很快就明白，应用架构组对宏的了解比我还少。至少我还和几个宏开发人员和Excel 老前辈谈过，知道实际上大家是如何用Excel Basic的，不外就是每天重新计算某张电子表 格，或者依照某个模式重新安排数据等等。不过应用架构组认为宏只是个学校习题，也不知 道实际上大家都是用宏在做什么。其中一位感受到压力，就提了一个想法：既然Excel已经 有底线和双底线功能，可能会有人想写一个宏去做三底线。是啊，真低级。于是我开始尽可 能的婉转地忽视他们。

这似乎惹恼了应用架构组的头头Greg Whitten。Greg好像是微软员工代码六号的元老。 他从盘古开天起就在这里了；没有人能明确指出他做过什么事，不过他常常跟比尔盖茨吃午 饭，而GW-BASIC是依他的名字命名的。Greg召开了一个「大」会议，开始抱怨Excel团队(指 我啦)如何恶搞宏策略。我们逼他提出明确的原因，不过他的论点就是没有说服力。我这个 大学刚毕业的菜鸟和六号老员工争论而且显然赢了，感觉真是不错。（你能想象这种事会在 那种穿灰法兰绒西装的公司发生吗？）当然很重要的原因是程序团队（由现在已经是微软副 总的Ben Waldman带领)的全面支持，因为程序团队负责写程序，所以他们对于做事的方法有 最终的决定权。

我很乐意让事情到此为止。不过如果应用架构组需要人关心想再吵架也没关系。他们想要我就会奉陪，只要能让程序员专心做事就好了。不过后来发生的事情更有意思，而且让我 很感动。我跟几个同事在Redmond的晴空下吃午饭，Pete Higgins朝我走过来。当时Pete是 Office部门的总经理。我当然知道他是谁，不过不认为他会了解我。

「约耳，怎么回事？」他说：「我听说你跟应用程序组有点过节。」

「哦，没有啦。」我回答：「我都可以搞定。」

「不用多说，我了解了。」他说完就走了。第二天就有谣言传回来：应用架构组解散了。 还不只这样，小组的成员都分散到微软不同的部门，而且离得很远。我再也没有听到他们的 事了。

我当然是非常的感动。在微软这家公司里，如果你是Excel宏策略的项目经理，即使来 公司还不到半年也不要紧，你就是Excel宏策略之神，连员工代码六号的老员工也不能挡你 的路。事情就是这样。

这件事传递了一个非常强烈的讯息。比如说它让大家对自己的工作更有责任感。也不能 再用「管理阶层批准了规格」这种想法来当借口，因为管理阶层并不会仔细看他们的规格。 管理阶层只管雇用聪明人并安排工作给这些人做。另一个意义是表示这里是个绝佳的工作场 所。有谁不想成为所属领域的王者呢？软件本质上就很容易分割成小块小块的组件，所以总 是可能分割出人员间的责任，让每个人拥有一个领域。这可能正是软件人喜欢在微软做事的 原因。

过了几年我换去Juno工作。这是一个在线服务和免费电邮的供货商。这次的经验和在微 软工作时正好相反。我底下有两个程序员对我负责，可是我的直属主管却一直在破坏我的(很 少的)权力，他跳过我直接分派工作给我下属，而且常常没通知我。连休几天假这种小事， 他都认为应该由他决定准假与否。

进Juno几年后，我去处理新的用户登入功能。我要负责翻修整个Juno 3.x(—个重要的 改版)的登入流程。当时我已经是技术团队里相当资深的成员了；我的考绩很好而且主管们 似乎也欣赏我的工作成果。不过他们就是没办法信任我。大概是命令与控制那套老想法的缘 故吧。

登入过程中有个地方要求使用者输入生日。Juno的登入程序很长，大约有30个画面，要 使用者填收入、喜好运动、有几个小孩、年纪多大还有其他一百个左右的问题。生日只是其 中很小的一部份。为了想让登入程序再简单一点点，我想把生日栏改成不限格式，这样就可 以输入"8/12/74"或"August 12, 1974"或"12 Aug 74"或其他内容。（你曾经用过Outlook 吗？这就像Outlook—样，随便打什么日期它都可以接受）。

我的主管并没有深入了解就决定他不喜欢这个作法。对他来说这变成自尊心问题。他先 吼了处理这个画面的设计师(根本没先告诉我)又再骂我。然后每天都提醒我一定要把它改成 他要的样子。接下来他找公司执行长来评估，还开批斗大会让执行长批斗我的新设计。事实 上连Juno的执行长都非常乐意干预公司最底层完成的工作。真是一脉相传的标准作业程序啊

我当然是很生气。这其实只是件小事，不过是喜好而已。某些人会喜欢我的作法，而某 些人喜欢他的。不管怎样讯息已经很明确的：你他X的照做就是了。这是非常命令与征服式 的心态，像是在比谁比较勇而不是在讨论使用接口设计。

我并不是说这是我离开Juno的原因，不过它的确解释了我离开Juno的理由：不管你多么 努力工作又有多聪明，也不管你是否在「负责」，你对再微小不过的事都没有权力。把你该 死的点子训练以及聪明睿智放一边，把那些让我们付钱请你的一切东西通通都丢到一边吧。 而且Juno的经理很多，大概占总员工四分之一，所以他们有的是时间可以到处乱伸手指，确 定他们能掌控。在微软却是副总自己从九号大楼出来找你，确定你有权力能把事做完。这对 比还真是强烈啊。

就某种程度而言，无能至极的管理流程是Juno之所以是一个纽约市公司而非西岸公司的 因素之一，所以还不太能深入现代的管理风格。这也是Juno经理人经验极度不足所导致的问 题，而源头则是在最上层，一位只待过D. E. Shaw的29岁执行长。他会干预每件能插手的大 小事，连出状况时错误讯息的字句也要管;这位执行长会定期大声责骂敢质疑其睿智的部属， 然后部属再把气出在程序员身上，程序员只好回家踢狗发泄。相较之下，在微软事情都是由 最底层完成，大多数经理的工作好像只是到处闲逛，把家具搬开不要挡到路，好让大家可以 专心工作。

三十三、巨无霸麦当劳与天才厨师JamieOliver

真是不可思议，为什么某些全世界最大的IT顾问公司会做出最糟的成果？

为什么那些最酷的暴发顾问公司刚开始时都有一连串成功的事绩，成长的非常快，然后却急速化为平凡？我曾经想个这个问题，也思考Fog Creek Software(我自己的公司)应该要如何成长。而我能 找到最好的教训是从麦当劳来的。是的，我就是在说那个可怕的汉堡连锁店。

大麦克的秘密在于它们并不是怎么好，不过每个汉堡不怎么好的程度是完全一样的。如果你愿意忍受不怎么好的食物，吃大麦克时绝对不会感觉任何丝毫意外的。

大麦克的另一个秘密是即使智商介「白痴」到「低能」之间，还是能生产出和全世界其他大麦克一模一样的汉堡。因为麦当劳超大本的操作手册才是它真正的秘方。手册里不厌其详的说明每个授权店制作大麦克时必须确实遵循的步骤。如果大麦克的汉堡肉在阿拉斯加的安克雷奇要煎37秒，那么在新加坡也会煎37秒，不是36秒也不是38秒。要做大麦克就是得遵循这该死的规则。

这套规则是由一些相当聪明的人(在麦当劳的汉堡大学)仔细的设计，让笨蛋能和聪明人一样照着做。事实上这套规则里有各种的防呆设计（比如薯条在油里炸太久会有铃声），就是要补 偿人的弱点。到处都有马表和各种计时装置。还有一个方法是要确保管理人每半小时会检查 厕所是否清洁。（提示：他们不会的。）

这个制度基本上假设每个人都会犯一堆错误，而做出来的汉堡必须始终如一，而你单点汉堡时总是会被问到要不要加点薯条。

纯粹好玩，让我们比较麦当劳厨师(完全不懂食物只会确实循遵规则)和英国原味主厨Jam i e Ol ive「那种天才。如果你选择现在离开本站，点那个连结去看原味主厨罗勒蒜泥蛋黄酱的MTV 风格短片，我绝对赞成并祝身体健康。毕竟拿麦当劳和美食主厨相比实在是荒谬无比，不过 请暂时压下怀疑，因为这里面有些道理可学。

原味主厨并不照什么乱七八糟的操作手册，也不会量任何东西。在他做菜的时候，你会看到各种食材到处乱飞。「我们只要多放一点点迷迭香，不会有事的，然后好好的摇一摇，」他说：「把它压碎。真完美。然后洒在上面就好了。」（是啊，看起来他的确只是随便洒上去。 问题是我来洒的话就是不行。）只要14秒左右，他就即兴创作了一道美食，海鲈片塞香草放 在蘑药和马铃薯上烤再加一点sa lsa-ve「de。真是美味。

嗯，我认为原味主厨的食物显然比麦当劳的东西好很多。虽然这听起来像个笨问题，不过还是值得花一分钟来问为什么。这问题并没有那么笨。为什么一家有着规模极巨又有庞大资源的大公司，可以获得最佳食材，现金多到不行，却无法生产出一道好菜？

想象一下原味主厨「电视节目」做腻了改去开餐厅。当然啦，由于他是个名主厨菜又很棒，所以餐厅门庭若市，钱好赚得不得了。

当你拥有一家很赚钱的餐厅，很快的就会了解即使每天都客满，即使一道前菜开价19美元而一杯可乐开价3.95,利润还是有限。因为一个主厨就只能做出这么多菜。所以得再请另一个主厨，或许再多开几家分店，或许还可以去其他城市开。

现在渐渐浮现一个问题，我们在技术业界称为可调性(scalability)问题。当你想要复制一 个餐厅，必须决定是要另外请一个能力与你相当的大主厨，还是要找一个没那么好但也没那 么贵的年轻厨师。不过大主厨师可能会想自己多赚一点，没必要替人打工;请年轻厨师的话， 老顾客很快就会发现菜变差了，以后也不会再去分店了。

处理可调性问题的一般作法是雇用什么都不懂的年轻厨师，然后让他照着精确到绝不出错的规则做菜。只要照着这些规则做，你也可以做出好菜！

问题是这方法不太行得通。好厨师即兴创作所牵涉的事非常多。好的厨师在农市看到上等的芒果，就会即兴做一道胡荽沙沙酱配当天的鱼料理。马铃薯不够时也会用芋头片顶替。只 会跟着指令的机械式主厨或许能在一切正常时生产出指定的菜肴，不过由于缺乏真正的才能 和技巧，不可能真的即兴创作。所以在麦当劳绝对不会有球根牵牛的色拉。

麦当劳需要一种特定品种的马铃薯。他们在世界各地都有种这种马铃薯，还会照规格切好大量冷冻储存以免短缺。预切及冷冻作业表示这些炸薯条不如新鲜的好吃，不过却保证始终如一也不需要大厨手艺。事实上麦当劳有数百个预备作业，确保任何厨房里请得到的笨蛋都能生产出质量一致的产品，不过是「稍微差一点」的质量。

到目前为止的总结：

1. 有些事需要天份才做得好。
2. 天份很难复制。
3. 有人尝试复制天份，要有天份的人建立规则让普通人照着做。
4. 结果所得到的产品质量很低。

你在IT顾问业可以看到完全一样的情节。下面这个故事你听过多少遍了？

麦克不太高兴。他请了一个大型IT顾问公司来建立系统。他请的IT顾问都很无能，只会一直讲「方法论(Methdology)」，花了几百万美元还做不好一样东西。还好麦克找到一位真正 聪明又有天份的年轻程序员。这个年轻人只拿了 20美元和一块披萨，一天内就把整个系统架 好了。麦克高兴得不得了，就把年轻的程序员推荐给所有朋友。年轻的程序员开始大赚银 子。不久工作就多到做不完，所以请了一堆人来帮忙。厉害的人股票选择权要太多了，所以 他决定请些更年轻刚毕业的程序员来「训练」六星期。问题是「训练」并不真的能产生一 致的结果，所以年轻的程序员就开始建立规则和流程希望产生更一致的结果。几年下来规则 手册愈来愈厚，不久就变成后一套叫做「方法论」的六册巨书。过了几年，当初的年轻程 序员现在已经成为巨大无能的IT顾问，带着一本「方法论」和一堆盲从方法论的手下。即使 方法论行不通的时候这些人还是硬干，因为他们根本不知道其他方法。何况他们也并不是真 有才能的程序员，只是受过六星期训练，好心的政治学科系毕业生而已。而这个新的巨大 无能IT顾问公司也开始陷入困境了。他们的客户都很不高兴。于是又出现另一个新兴的天才 程序员把客户都抢光，然后整个循环又重来一遍。

我不需要在这里指名道姓，这个循环已经发生过很多遍。所有IT服务公司都会贪心，希望成长的速度能超过找到优秀人员的速度，于是发展一层又一层的规则和流程来，帮助制作出「一 致」但算不上一流的成果。

不过规则和流程只有在一切正常时行得通。前几年到处都出现各种「数据库(data-backed) 网站」顾问公司，拿些「建立数据库网站的十四项须知」之类的东西教出一堆生手来填公司 职缺。（「小朋友，这个叫select叙述，去架个网站吧」）。不过现在网络泡沫破灭，而高 阶GUI程序设计和C++技术还有真正的计算机科学需求突然起来，这些口袋里只有select叙述 的小伙子就面临严峻的学习曲线，完全无法跟上。不过他们还是一直照着第17章数据库正 规化的方法在试，不过突然间这些方法已经不适合这个新世界了。这些公司出色的总V办乂 应该都能适应新世界，毕竟他们都是有才能的计算机科学家，什么东西都学得会。不过他//7 展建立你公司是变不了的，因为公司用规则手册取替了才能，而规则手册无法适应新时代。

这个故事的教训是？小心方法论。方法论可以让每个人的表现都提升到不佳但可接受的程度， 不过同时也会产生很多约束而激怒更多的聪明人。就我而言，有才华的厨师显然不会乐于在 麦当劳做汉堡，廢證正是麦当劳的规则。是否如此IT顾问才如此吹墟他们的方法论呢？（来 打我啊。）

这对Fog Creek有什么意义？嗯，我们从来没有把成为大顾问公司视为目标。我们一开始就把顾问工作视作一种手段，真正的长程目标是成为一直有利润的软件公司，而我们得做一些顾问工作来补软件收入的不足。经过几年的经营，我们的软件收益已经成长到相当程度，顾问工作相对成为分心的低收益业务，所以现在只做对我们的软件有直接帮助的顾问合约。 你也知道，软件#常崩容易复制。每多一个人购买FogBUGZ，我们就不用花任何成本就能多 赚到一些钱。

更重要的是我们坚持雇用最好的人，如果找不到够好的人，我们很乐意维持小局面(不过一 年有六周休假，找人似乎不构成问题）。而且在找进来的人还不足以成为教导新人之前，我 们拒绝再扩充。

三十四、没有什么像IT看起来那么简单

问题是这样的：你可以用选单由网络上汇入档案（Import Web Page)，也可以用 鼠标拖拉磁盘里的档案汇入CityDesk。不过没有选单可以汇入磁盘中的档案，所 以大家不是没有发现可以这样做，就是尝试用Import Web Page功能汇入磁盘中 的档案，而后者当然也是不能用的。

我认为用两页的精灵可以轻易的解决这个问题。大概的方法是用精灵的第一页问 「你要由哪里汇入数据？」，如果你选「磁盘」，第二页就提示你输入档名；如 果选择「web」，第二页就提不输入URL路径。

我几乎要开始实作，不过还是停下来着手写一份迷你规格。下面是这份规格的全文：

第一页 你要由哪里汇入数据？（磁盘/Web)

第二页(磁盘） 标准的开启档案对话盒第二页(Web) 有迷你网页浏览器的URL输入画面

突然间我想到一件事。有办法把通常由0S提供的开档对话盒激在精灵里吗？

呃…

我去查过了。结果是可以的，不过不太好玩而且要花几个小时工夫。那么可以不 用精灵来做吗？我把规格重写成：

两个选单： 1)由Internet汇入网页-> 显不URL对话盒 2)由磁盘汇入网页-> 显示开文件对话盒

好多了。三分钟的设计省了我几小时的程序工夫。

如果你这辈子写过20分钟以上的程序，可能已经发现这个很好的基本原则：没有事情像表面看起来那么简单.

就连复制一个档案这么简单的动作都充满陷阱。如果第一个参数是目录会如何？ 如果第二个参数是档案又如何？如果目的地已经有相同档名的档案又会怎样？

万一没有写入权限又怎么办？

如果档案拷到一半出错要如何？如果目的地是需要认证才能继续的远程机器又 如何？如果档案很大而联机很慢，必须显示进度栏时又会如何？万一传输速度慢 到刀手完全不动…你要等多久才放弃并传回错误讯息呢？

有个好方法可以用来面试应征测试工作的人，就是要他针对一件简单的操作，把 所有可能出错的地方全都列举出来。微软对面试测试人员有个经典题目：如何测 试开档对话盒？优秀的测试人员可以滔滔不绝地列出几十种测试项目（在你选好 档案，要按「开启」之前，另一个使用者把原本存在的档案杀掉了）。

好了，所以我们得到一个定理：没有事情像表面看起来那么简单。

软件工程里有另外一个定理，就是随时随地都要尝试降低风险。其中一项必须避 免的重大风险是进度风险，就是某件事花的时间比预期多。进度风险的坏处在 于你会被老板吼而很痛苦。如果你觉得这没什么，可以改由经济观点考虑进度风 险的坏处。如果因为当初评估需要一周而决定要做某个功能，事后才发现其实 需要20周，那么前面的决策就错了。如果你早知道需要20周很可能就会做不一样 的决定。而错误的决策愈多，印有你公司标志的购物袋愈有可能流落到资产清算 的仓库，而同时间你的前执行长还在怨叹「真惨啊，我们还没有被烂公司报导前 就倒了！」

「没有事情像表面看起来那么简单」再加上「降低风险」只会导出一个结论：

你必须先做设计再去实作。

我很抱歉让你失望了。没错，我知道你看过Kent Beck的书，而且现在认为不先 设计而直接实作是正常的。很抱歉，这并不正常。改程序就是没法子和改设计 文件「一样简单」。大家一直都这样说，不过这并不对。「现在我们都用Java 和XML这种高阶工具，几分钟就可以改好程序，为什么不直接在写程序时做设计 呢？」我的朋友，你可以替你老妈装上轮子，不过她并不会因而变成巴士。另 外如果你自认能把错用线程的档案复制函数，重整改正成先占多任务的作法，而 且速度和我写这个句子一样快，你根本就在自己骗自己(deep denial)。

无论如何，我不认为极限程序设计真的在提倡零设计。他们只是说「不要做无谓 的设计」，而这一点是没有问题的。不过大家所至如勺不是这么回事。大部份程 序员都尽可能地找寻能不在实作前先做基本设计的借口，所以他们就像捕蚊灯里 的蚊子一样黏住「无设计」这个想法。这其实只是偷鸡不着蚀把米的另类懒惰。 我懒得先在纸上设计功能所以直接写程序，程序做得不对只好再花时间修正，结 果只会花更多的时间。还有更常见的状况，就是程序写错了可是来不及改正， 于是产品的质量低落，只好一直找借口解释东西为什么「必须写成这样」。根本就是懒散又不专业。

当Linus Torvalds痛斥设计时谈的是巨大的系统，这种系统必须逐渐演进，否 则就会像Multics那样死路一条。他说的并不是你的档案复制程序。人们认为 Linus有很清楚的计划，确实知道往后要怎么进行，无怪Linus会不怎么看重设 计。所以别上当了，这种作法很可能并不适合你。何况Linus比我们要聪明多， 他能做的事我们一般人是做不来的。

渐进式的设计和实作是好的。常常发行也是好的。不过就软件包或大众市场而言， 频繁的发行新版会让客户疯掉，并不是个好主意，应该改成在内部密集的发行。 太拘泥于设计细节会浪费时间，我从来没看过哪个项目，会因不需动脑的流程图 或UML或CRC或是其他流行玩意而获益。Linus说的那些有一千万行程序的超大型 怪物系统应该要逐渐演进，因为人类确实不知道要如何设计那种规模的软件。

不过当你坐下来要写档案复制的程序，或是计划下一版软件的功能时，一定要做 设计。不要让那些说法迷惑了你。

你可以在四月29到五月一日间，到麻州剑桥的Cutter Summit和我当面讨论，我会和Tom DeMarco, Kent Beck和其他人一起。

三十五、提防非自主开发综合症

猜谜时间到了。

1.程序代码重用是:

a)好的 b)不好的

2.重新发明轮子是:

a)好的 b)不好的

3.非我发明(Not-Invented-Here)症是:

a)好的 b)不好的

当然啰，每个人都知道应该要善用别人的工作成果。所以正确答案当然就是1(a) 2(b) 3(b)。确定吗？

别说得那么快!

非我发明（NIH)被公认为典型的管理病状，指某个团队拒绝使用不是自己创造的 技术。NIH症的患者显然只是很小家子气，只因为没办法居功就拒绝为整体组织 的利益贡献(对吧？）。你可以在当地大书店的沈闷商业史区找到一堆故事，说愚 蠢的团队花了几百万美元和12年的时间，却只做出某些只要9.99元就能在 Egghead(译注：在线零售商）买到的东西。任何人只要有注意过去三十年间计算 机程序设计的进展，就会知道重用是所有现代计算机系统的圣杯。

没错，我也是这样想的。所以我在做第一版Visual Basic for Applications的 项目经理时，就很小心的联合了四个(注意是四个哦)微软内不同的团队，要做出 Excel VBA里的定制对话盒。整个想法非常复杂而且各部门间相互牵连。有个叫 AFX的团体在制作某种对话盒编辑器。然后我们会利用OLE团体新写的程序把一个 应用程序嵌入在另一个程序里。而Visual Basic团队则是提供里面的程序语言。 经过几星期的协商，AFX和OLE还有VB团队原则上都同意这个作法。

我去到Andrew Kwatinetz的办公室。他当时是我的经理而且我知道的一切都是他 教的。「Excel开发团队绝对不会接受这个方法，」他说：「你知道他们的格言 吗？『找出相关性，然后去掉。』他们绝对不会接受这么不独立的东西。」

真-有-趣-啊。这事我竟然不知道。我想这解释了Excel团队有自己的C编译程序的 原因了。

现在我确定很多读者已经笑到在地上打滚。「微软真是笨啊，」你会这样想「他 们拒绝用别人的程序代码，而且只是为了一个产品还自己做编译程序。」

别说得这么快，老弟！ Excel团体粗鲁的独立心态意味着他们总是能如期上市， 而且出来的程序代码都有一致的高水平。另外他们的编译程序（由80年代开始的) 会产生pcode，所以不必修改就能在Intel计算机和麦金塔68000芯片的系统上执 行。pcode也使得执行档在Intel平台上只有原本的一半大小，所以由软磁加载时 能更快启动而且需要较少的内存。

「找出相关性，然后去掉。」当你在一个非常非常优秀，有着伟大程序员的团队 工作时，其他人的程序坦白说都是有虫的垃圾，何况其他人都没法子准时上市。 如果你是法国蓝带学院的主厨，需要新鲜熏衣草时你会自己种而不是去菜市场买， 因为有时候新鲜的熏衣草可能会缺货，有时候店家可能拿不新鲜的熏衣草假装新 鲜卖给你。

事实上最近的网络热潮中，一堆装内行的企业作家说未来的公司会完全虚拟化， 公司变成只有几个时髦的家伙待在起居室里喝着白葡萄酒，所有事情都外包出去 做。这些换气过度的「梦想家」并没有注意到市场机制是为所增加的价值而付钱 的。一个客厅加上两个雅痞，向A公司买一套电子商务引擎，拿B公司做的货来卖， 再找C公司处理库存和送货，客户服务则交给D公司，这种作法实在没有增加什么 价值。事实上如果你曾经把很关键的业务拿去外包，就会了解外包其实是个地狱。 不能直接控制客户服务，客服就会烂到不象话，烂到像某人在网志上写的，想在 电话公司找个人(任何人都好)去做些最最简单的事都办不到。如果物流外包出去，你的物流包商或许对实时送达会有不同的认知，而你的客户就会不太高兴，可是 你一点办法都没有，因为重新找一个物流包商要花三个月。何况事实上你根本不 会知道客户不高兴，因为他们找不到你，因为你设立了一个外包的客户服务中心， 宗旨就是不要让自己听到客户的声音。至于你买的电子商务引擎？绝对不可能像 Amazon用obidos做的那么有弹性，obidos可是人家自己写的（否则Amazon跟买那 种东西的竞争者相比就没有优势了）。另外也没有现成的web服务器在速度上比得 过Google自己亲手制作并优化的服务器。

很不幸的这个原则似乎与「重用程序代码好，重新发明轮子坏」的想法直接冲突。 我能给的最好建议是：

如果是核心的事业功能，不管是什么都要自己来做。

找出你的核心事业能力和目标，然后在公司内执行。如果是家软件公司，写出优 越的程序代码就是成功的方法。公司餐厅和光盘压制都可以外包出去。如果是家 药厂，就去写药物研究的软件，可是不要写自己的会计软件。如果是家网络会计 服务公司就写自己的会计软件，不过别想做杂志广告。如果有客户的话就千万不 要把客户服务外包出去。

如果你在开发计算机游戏，而且地图(plot)部份是你的竞争优势，大可去用其他 厂商提供的3D链接库。不过如果游戏特色是很酷的3D特效，3D程序部份最好自己 来。

我怀疑这个规则唯一的例外，就是当你自己的人完全比不上别人，不管自己 做什么都会失败。没错，这种地方很多。如果是这种情况，我就帮不上忙了。

三十六、策略I：BEN&JERRY公司与AMAZON

正在建立一家公司吗？那你得做一个很重要的决定，因为这个决定会影响其他所 有的事情。不管做了什么其他事，你绝对一定要找出自己属于哪个阵营，然后一 切都照着进行，否则就会大难临头。

是什么决定呢？是要有组织有利润地慢慢长大，还是要找很多资本像大爆炸一样 快速成长。有组织的模式是由小开始，长时间针对几个目标慢慢建立一个事业。我称之为Ben and Jerry模式，因为这家公司（译注：传奇性成长的美国冰淇淋公司）非常符合 这个模式。

另一个模式通常被称为「快速范壮(Get Big Fast)」（亦名「抢占地盘(Land Grab)」），需要募集很大的资本，然后不考虑获利只是尽可能的快快长大。我称 之为Amazon模式，因为Amazon的创办人Jeff Bezos已经成为快速范壮模式实质 的代言名人了。

让我们看看这两种模式间的某些差异。首先要问一个问题：你想要做有竞争的产 业还是不竞争的产业？

Ben and Jerry 模式	Amazon模式
很多原有的竞争者	新科技，一开始没有竞争者

如果你像Amazon—样没有真正的竞争者，就有机会能成功「抢占地盘」，也就是 说尽快获得更多的客户，提高后来竞争者的竞争障碍。不过如果你进入的产业已 经有很多稳定的竞争者，抢地盘的做法就说不通了。你必须把客户由竞争者那边 拉过来才能建立自己的客户群。

一般说来，创投界都不太热衷于进入有棘手竞争者的产业。就我个人而言，其实 并不太怕与现有的公司竞争；或许因为我在微软Excel工作时期，看到了 Excel 完全打败原本一统江湖的Lotus 123。排名第一的字处理器Word取代了 WordPerfect，而WordPerfect之前又取代了WordSta「，这些软件都曾经独占市场。 虽然在Ben and Jerry出现之前并不是没冰淇淋吃，可是它还是能成长为一个难 以置信的事业。所以说如果真想的话，要取代竞争者并非不可能(我会在以后的 策略信里谈谈可行的方法）.

另一个取代竞争者的问题和网络效应及锁定（lock-in)有关：

Ben and Jerry 模式	Amazon模式
没有网络效应：客户锁定薄弱	网络效应机客户锁定都很强

「网络效应」是指现有的客户愈多，就会获得愈多的新客户。这个现象是基于 Metcalfe's Law：网络的价值等于用户数目的平方。

eBay就是个好例子。如果你想把你的旧百达翡丽表卖掉，去eBay可以出到较高的 价格，因为那里的买家比较多。如果想买支百达翡丽表也会去eBay，因为那里的 卖家也较多。

另一个非常强的效应是ICQ或AOL I nstant Messenge「等专属的聊天系统。如果你 想和别人聊天就得去他们的地方，而目前ICQ和AOL的人最多。有可能你的朋友在 用这些服务之一，而不是MSN Instant Messenge「之类较小的服务。即使用尽微 软的力量，金钱和营销手段，也没法子切入拍卖或即使通讯产业，因为网络效应 实在太强了。

「锁定」是企业有某些东西让客户不想更换。即使所用的Internet服务并不十分 的好，可是还是没有人会想更换，因为要换电邮地址还要把新的电邮地址告诉大 家，实在太麻烦了。另外如果新的字处理器不能读旧档案，大家也都不会想换字 处理器。

我称之为秘密徵定(stea/t7 lock-in)的潜伏版比单纯的锁定效果更好，这种服 务在你根本不知道的状况下就锁定你了。举例来说，像PayMyBills.com这种新 服务会帮你收账单并扫描进计算机，让你可以在Internet上看。它们通常会提供 三个月的免费服务，不过如果三个月后你不想继续用，就只好自己逐一联络各 个账单寄发单位，请他们把寄送地址改回你家。这个痛苦繁琐的动作或许会让你 不想换掉PayMyBills.com，只好让他们继续每个月由你的银行账户吸走8.95美 元。明白了吧！ 如果你要进入一个有网络效应和锁定特性的产业，又没有已稳定的竞争者，那么 最义/采用Amazon模式。否则等其他人先下手，你就什么都分不到了。

来个简短的案例研究。1998年A0L砸了大钱以每五周增加一百万名用户的速度成 长，他们有些很好的功能有秘密锁定的效果，比如聊天室和实时通讯功能。当你 找到一群想聊天的朋友，根本不会换Internet供货商，因为换供货商等于是要找 一群全新的朋友一样。外面一个月十元的Internet供货商到处都是，A0L却敢一 个月收约22美元，我个人认为这就是主要原因。

当我在Juno工作的时候，管理阶层就是没法子认清这一点，也没有把握机会在大 家都开始上网的抢地盘时期打败A0L。由于害怕再增资会稀释原股东的股权，所 以他们并没有砸足够的钱去抢客户。另外他们也没有从策略面去考虑聊天室和实 时通讯，因此从未开发像像A0L—样具秘密锁定效果的软件功能。现在Juno有大 约三百万名用户，每个用户平均一个月付5.50元，而A0L有两千一百万用户，平 均月费则是17元。差真多啊！

Ben and Jerry 模式	Amazon模式
只要小额资金：可快速达损益两平	需要极大量的资金：要很多年才能获利

Ben and Jerry型的公司会靠某人的信用卡起家。在刚开始的几个月或几年内， 必须采用能极快获利的商业模式，而这可能并不是他们希望最后能达到的模式。 举例来说，你可以，想成为一个年营业额两亿的大型冰淇淋公司，不过现在只能存 就先在Vermont开一家小冰淇淋店，希望能靠它赚钱。如果真有盈余的话再投入， 稳定地扩充事业。Ben and Jerry的企业史说他们刚开始投入一万两千美兀。 ArsDiqita则说他们刚开始有一万一千元的资本。这些数字听起来就像是 MasterCard卡的信用额度。

Amazon型的公司募资的速度比任何人花钱都快。这是有原因的，因为他们要和时 间竞赛。如果他们身处一个有网络效应却没有竞争者的产业，最好能以极速成长， 因为这时候每一天都有关系。另外有浪多方法可以用钱换时间(参阅附栏)，而且 几乎每种方法都很好玩。Ben and Jerry型的公司根本付不起这种钱，所以就只 能接受慢慢成长。

Ben and Jerry 模式	Amazon模式
企业文化很重要	不可能有企业文化

当你每年成长超过一倍，前辈根本不可能把企业价值传授给新进人员。如果某位 程序员晋升成经理，突然要带五个昨天新雇的手下，绝对不可能有多少指导。 Netscape就是最夸张的例子，在一年间由五个人扩充到两千人，结果企业文化就 变成大杂烩，每个人对公司的价值认知都不一样，朝着四面八方互相拉扯。

对某于公司或许不算什么。不过对其他公司来说，企业文化是公司存在很重要的 理由。Ben and Jerry是因为创办人的价值而存’在的，他们不容许公司成长的速 度超过公司文化散播的速度。

让我举一个虚构的软件作例子。假设你想切入字处理器的市场。虽然这个市场似 乎已经被微软包了，不过你看到一个利基，某些人用字处理器时是绝对不能当掉 的。你打算做一个超级稳定又专业，绝不当机的字处理器，打算当高级品卖给只 靠字处理器维生的人。（是啦，这很夸张。不过我说过这是个虚构的例子了）.

于是你的企业文化可能包括各种撰写超稳定程序的技巧：单元测试，正规测试， 正规程序代码审查，编码规范，大型的QA部门等等。这些技巧并不单纯，都要相 当时间才学得会。新程序员学习如何撰写稳固的程序时，必须要有经验的人教导 并训练。

当企业成长太快无法进行这种教导及训练时，就无法再传递这些价值。新员工没 有提升水平就会开始写出不可靠的程序。他们不会检查malloc()的回传值，程 序会在他们想不到的怪地方错掉，而且没人有时间审查他们的程序或教他们正确 的做法，于是你超越微软的竞争优势就整个崩溃了。

Ben and Jerry 模式	Amazon模式
错误成为珍贵的教训	错误很少会被注意到

一家成长太快的公司根本不会注意到自己犯了大错，特别是钱花太凶的那种错误。 Amazon用一亿八千万的股票并购了 Junglee这个比价购物服务，然后才突然发现 比价购物服务对公司的业务不太好，于是只好把它收起来不做。手上有大把的银 子，即使犯了愚蠢的错误也很容易掩饰

Ben and Jerry 模式	Amazon模式
需要很久才能变大	很快就可以变大

快速茁壮会产生成功的印康(即使不是真的）。潜在的员工看到你一周内请了30 个新人，会感觉自己是某家将要上市的振奋的成功大公司的一员。如果换成只有 12个人和一条狗的「懒洋洋小公司」，大概不会有一样的想法。虽然那个懒洋洋 的公司有赚钱而且以后可望成为更好的公司。

有个经验法则是这样的，你可以让大家有个好的工作环境，或者可以允诺大家以 后一定会发大财。不过一定得选一样，否则是请不到人的。

有些员工会对很可能上市而且给很多股票选择权的公司感兴趣。这些人即使每天 分分秒秒都很痛苦，还是会愿意投入三到四年在这种公司工作，因为他们看到彩 虹尽头的奖杯。

如果你有组织地慢慢成长，奖杯就变得很遥远。这时候只能让工作环境更好，让 旅程本身变成乐趣。不可以要人每周工作80小时。也不可以用那种塞满折迭桌 和硬木椅，又大又吵的办公室。你必须提供适当的休假。同事不能只是同事，还 要成为朋友。经理人得开明不可以对大家唠叨，不可以成为呆伯特里那种事必 躬亲的经理人。如果这些都做到了，就可以吸引很多被上市致富的梦骗过太多次 的人；这些人现在只想找一个还过淳去的工作。

Ben and Jerry 模式	Amazon模式
你可能会成功。不过一定不会损失太多钱。	成为亿万富翁的机会很低很低，而失败的机会很大。

采用Ben and Jerry模式的话，只要你够聪明就会成功。可能过程有点辛苦，有 时候状况好有时候很惨，不过除非出现另一次大庸条，否则一定不会损失太多钱， 因为一开始就没有投入多少资金..

Amazon模式的问题在于每个人都在想Amazon，可是Amazon只有一个。你必须想到 其他95%的公司都花了极大量的创投资金，然后却因为没人要买产品而失败。如 何你走Ben and Jerry模式的话，至少远在花完一张MasterCard卡的信用额度之 前，你就会知道没人要买你的产品。

你所能做最糟糕的事

你所能做最糟糕的事，就是无法决定自己要成为Ben and Jerry型的公司还是 Amazon型公司。

如果要进入一个有锁定和网络效应，可是目前没有竞争的市场，最乂/采用Amazon 模式，否则可能会和Wordsworth.com—样下场，这家公司比Amazon早两年开始可 是现在却默默无闻。还有更糟的情况，你会变成像MSN拍卖那么的幽灵网站，根 本不可能有机会打败ebay。（看看Wordsworth的回答）

如果你想进入一个现有的市场，想快速茁壮只会白白浪费钱，就像 BarnesandNoble.com—样。最好的期望是做些足以维持和有利可图的东西，这样 可以多撑几年慢慢胜过对手。

还是无法决定吗？可以考虑其他方面。想想看你个人的价值？你比较喜欢拥有想 Amazon还是像Ben and Jerry那样的公司？可以看这些企业的历史，比如Amazon 和Ben and Jerry的创立过程(虽然都在歌功颂德），看看哪一个和你的核心价值 相符。事实上微软是个更适合Ben

and Jerry型公司的范例，而且微软有很多史 料可参考。就某种程度来说，微软很「幸运」可以得到PC-D0S的生意，不过这家 公司一直都有赚钱而且一路成长，所以他们可以一直伺机等待突破的机会。

想想你的风险及报酬概况。即使机会比中乐透还低，你还是会想拼一下在35岁前 成为亿万富翁吗？那么Ben and Jerry型的公司就不适合你了。

或许你所能做最糟糕的事，就是决定自己要拥有一家Amazon型的公司，可以后来 却照Ben and Jerry模式经营（而且一直否认）。Amazon型的公司一定淳尽可能地 用钱换取时间。你可以坚持用市价来请程序员，还认为自己聪明会省钱。不过其 实并不聪明，因为这样会要六个月而不是两个月，而多用的四个月表示你错失了 圣诞节的购物季，所以现在等于用了一年，而且还会让你整个商业计划书都无法 实现。你可能认为除了Windows版之外再出个麦金塔版是很聪明的作法，其实会 用掉你两倍长的时间，因为程序员要多建立一个兼容层，另外也只会多15%的客 户。所以啰，其实并不怎么聪明，对吧？

两种模式都可以成功，不过你得挑一种然后坚持下去，否则会发现一切都莫明奇 妙地出问题，而且也不知道原因。

三十七、策略II：鸡与蛋问题

做广告就是要说谎但不能被抓到。大部份的公司做宣传活动，都只是把公司最差 的事实拿出来反过来讲（「说谎」），然后深入彻底的撒谎。让我们称之为「以反 复主张来证明」。举例来说，坐飞机旅行很狭又不舒服，空服员粗鲁又让人不快， 事实上整个航空裏统就是种折磨。所以几乎所有的航空公司广告都在说坐飞机是

多么的嚴而餘炔，整个过程中你又多么的被，棍殿。英国航空有个广告是一个商 人坐在飞机上梦到自己是个襁褓里的婴儿，看到这个广告时所有合理性荡然无存。

要举别的例子吗？造纸公司一直在大量砍伐我们国家的森林，把根本不属于他们 的老树林砍得干干净净。所以他们做广告时当然就会展现漂亮的老松林，然后说 他们有多么在意此环境。香烟会害死人，所以烟广告就展示生命，比如拍健康的 人微笑着在户外运动。如此类推。

麦金塔第刚问世时根本没有软件可以用。理所当然的苹果公司就编了一大本光面 目录列出所有「可用」的伟大软件。不过有一半的项目用小字印着「开发中」， 而剩下一半是不管怎么都拿不到的。其中有些产品烂到不会有人买。不过即使有 了这样一本厚厚的光面目录，即使里面每一页都有一个用美丽说明包装的软件 「产品」，还是无法掩饰你的128KB麦金塔买不到字处理器或电子表格的事实。 NeXT和BeOS都有类似的「软件产品指南」。（NeXT和BeOS的盲目崇拜者注意：我 不需要任何有关你这些该死的操作系统的鬼扯，好吗？要的话在你自己的专栏里 写。）软件产品指南唯一的意思就是该系统没有软件可用。当你看到这些巨兽时， 朝反方向全速逃离吧。

Amiga、Atari ST、Gem、IBM TopView、NeXT、BeOS、Windows CE、General Magic，一长串失败的「新平台」不断出现。由于它们是f台，照字面上如果没有好软件 就不会太有趣。虽然有极少数的例外(我相信像Am i ga或RSTS-11这种神秘而冷门 平台的无聊支持者会寄一大准电邮给我)，稍有常识的软件开发者在景气好时， 都不会刻意为有十万用户的平台写软件，因为花同样的工夫可以为Windows这种 有上亿用户的平台写软件。依旧有人替这些怪胎系统写软件，证明了利益并不 是唯一的动机，而宗教狂热仍然长存。亲爱的，祝你好运。你替Timex sinclair 1000写了一个不错的microEmacs复制品。做得好！给你两毛五，去买颗糖果吧。

所以说如果你身在平台创造的产业，可能会面临一般称之为湾生蛋蛋生湾游柯题。 除非拥有好的软件可用，否则没人会买你的平台，可是你的平台要有很多用户， 才会有人替你写软件。痛苦啊！这就有点像是一个哥帝安结(Gordian Knot)，不 过说是哥帝安死亡漩涡可能更贴切。

鸡生蛋蛋生鸡问题者以及类似状况是定策略时不可不知的最重要元素。嗯，是啦， 你不知道或许也还是能生存：事实上Steve Jobs在不管这个问题的状况下两度创 立了一个事:#。不过我们其他人并没有Jobs的个人现实扭曲场(Personal Reality Distortion Field)可用（译注：很

多人说Jobs有神奇的影响力，驱使别 人不顾现实为他卖命)，所以必须认真努力研究。

第一课。鸡生蛋蛋生鸡问题的典型范围是在软件平台。不过这里有另一个鸡生蛋 蛋生鸡问题：每个月翁百”万家信用卡公司邮寄出刀必份账单给客户。大家就会签 好书面支票塞进上兆的信封再寄回去。这些信封会装到大箱子里，运到人工便宜 的国家再开箱处理。不过整个过程相当花钱，我最后听到的数字是每张账单超过 一美元。

对我们这些Internet的聪明人来说真是个笑话啊。「把账单电邮给我，」你会说 「我要在线付款！」你还会说：「假设这样只要花大概十万分之一分钱。你可以 省下刀百:万」或是其他类似的话。

你说得没错。所以很多公司都想进入这个技术上称为嚴卓递交(Bill Presentment)的领域，而微软是其中之一(想不到吧）。他们的方案TransPoint是这样子的：提供一个网站让你可以上去看账单，然后你付款给他们。

所以现在如果你用这个微软的系统处理账单，每几天就得去该网站看看有没有新 账单来以免漏缴。假设你一个月有十张账单，这应该不算大麻烦。不过还是有别 的问题：只有少数商家会透过这个系统收帐，所以其他账单还是得用别的方法处 理。

最后的结果呢？就是不值得。这样的系统能找到十万人来用，我都会觉得很惊讶。 于是微软就必须去找商家商量：「请透过我们的系统来收帐！」然后商家会问： 「可以啊，价钱怎么算？」微软回答：「五毛钱！这比一美元便宜多啦！」于是 商家说「不错，还有其他费用吗？」微软就说啦：「哦，对了，你还得安装软 件并把你我的系统连接起来，再加上整个运作费用，大概要廿五万美元。」

由于微软这个系统的用户数量太少，实在无法想象谁会为了节省37个客户的五毛 钱而付廿五万美元。啊哈！鸡生蛋蛋生鸡问题悄悄出现了！除非商家参与否则 客户不会来，可是没有客户的话商家是不会参与的！最后微软只能花钱脱离困境。 不过较小的公司可没法子这么做。那么你能怎么办呢？

就鸡生蛋蛋生鸡问题的处理来说，软件平台其实可以给我们一些不错的提示。让 我们稍微看看由IBM-PC起这些年间，个人计算机软件平台的历史；或许能发现某 些东西！

很多人认为丨BM-PC需要用PC-D0S。这并不正确。IBM-PC刚出来时有三种操作系统 可以选择：PC-DOS、XENIX(—套8位的烂UNIX，是「微软」出品的），还有某个叫 UCSDP-System的东西(信不信由你，这东西就像Java:美好、缓慢、可携的 bytecode，还比Java早20年。)

如今大多数人都从未听说过XENIX或UCSD这些怪东西。你们这些现代的小朋友可 能会认为，这是因为微软用营销势力或其他手法夺取了小小的操作系统市场。 这绝对不是真的；那时候微软还很小，当时最有营销势力的公司是拥有多个操作 系统的Digital Research。那么为什么PC-DOS会是这三家竞赛的赢家呢？

在PC出现前能用的真正操作系统只有CP/M，不过用约需一万美元的CP/M计算机市 场太小了。这些计算机贵又不稳用，起来也不方便，不过还是有人买来当字处理 器用，因为CP/M上有一套相当好的字处理器WordSta「，而Apple I I根本不葳做字 处理(连小写字母都没有）。

现在有件已知的小小事实：连DOS 1.0的设计都内建了CP/M兼容模式。DOS不只有 一套很好的新程序接口（硬派程序员所知道的INT 21)，还完整地支持了原本的 CP/M程序接口。它刀手可以执行CP/M软件。事实上WordSta「的程序代码只改了一 个字劳就移植到DOS 了（真正的程序员可以告诉你是哪一个字节，不过我早就忘 了)。

这件事值得再说一次。把WordSta「移植到DOS时，程序代码只改了一个字貫。好 好想一下。

好啦。

想到没？

DOS是琢为一方:給#存软洋展才会流行。而它能有软件用是因为Tim Paterson想 到要提供CP/M兼容功能，因为在那个黑暗时代某人就有智慧去处理鸡生蛋蛋生鸡 问题。

快转一下。在整个PC平台的展史里，只发生过两次足以影响到每一个PC用户的重 大典范转移，我们都转移到Windows 3.x，然后又换到Windows 95。只有极少数 人转移到其他地方。微软阴谋要接收全世界吗？很好，你大可那样子想。不过我 想到另一个更有意思，可以回归到鸡生蛋蛋生鸡问题的原因。

我们都要转移至Wndows 3.X。这句话里的3是个重要的线索。我们大家为什么不是 都转换到Windows 1.0 或Windows 2.0?又或后来的 Windows 286或Windows 386 呢？因为微软发行了五个版本才终于「做得好」吗？错。

真正的原因远比这个更微妙，它跟Windows 3.0所需的Intel 80386芯片中一个很 神秘的硬件功能有关。

1. 功能一：旧的DOS程序在显示画面时是直接写入文字字符在屏幕上 对应的内存位置。要让程序输出快速而专业，这是唯一的办法。不 过Windows是以绘图模式执行。使用旧的Intel芯片时，微软的工 程师在执行DOS程序时没有其他选择，只能切到全屏幕模式执行。 不过用80386就可以设定虚拟内存区块和中断，让在程序试图写入 显不内存时通知操作系统，然后Windows就可以立即在图形窗口绘 出相同的文字。
2. 功能二：旧的DOS程序假设自己拥有整颗芯片，所以通常不能共存。 不过Intel 80386有能力创造多个「虚拟」PC，而且每一个就和一 颗完整的8086—样，所以虽然还有其他程序一起执行，可是旧PC 程序还是能假装自己独占整台计算机。

所以用Intel 80386上运行的Windows 3.x是第一个能完整执行多个DOS程序的版本(就技术上Windows 386也可以，不过在W i ndows 3.0问世前80386数量少又贵）。 Windows 3.0是第一个真正能执行你所有旧软件的版本。

Windows 95?也没问题。它有很好的32位新API，不过还是能正常执行旧的16位 程序。微软非常重视这件事，花了很多工夫Windows 95在上测试所有找得到的旧 程序。Jon Ross是Windows 3.x版SimCity的作者，他告诉我他不小心在SimCity 里留了一只虫，会去读取刚释放掉的内存。是的，这在Windows 3.x是可以的， 因为内存不会跑掉。接下来的事非常神奇：SimCity在Windows 95的beta版上测 试有问题，微软追查到这个问题然后在W/ndws 95里加一段找寻S/mC/ty游涯淨 代码。如果发现SimCity正在执行，内存配置程序就会进入特别模式，这时并不 会马上释放内存。就是这种向后兼容的坚持让大家愿意升级到Windows 95。

要如何破解鸡生蛋蛋生鸡问题，你应该开始有些概念了吧。就是用一个向后兼容 模式，依你的需要提供一卡车的鸡或蛋，然后坐下来大赚钞票。

哈!现在回到账单递交的事情。还记得账单递交吧？这个鸡生蛋蛋生鸡问题是说， 这个服务只能替你处理少部份的账单，所以你不会去用。要怎么解决呢？微软没 有想出来，不过PayMyBills.com和其他几家硅谷新创公司都同时找到答案。提供 一个应后砉容模式：如果商家不支持这个系统，就请他们把可恶的书面账单寄到 公司在加州的地址，再找一堆人来拆信封扫描进计算机，现在你所存的账单都 在他们的网站上了。由于世界上所有商家都可以用这个系统，客户可以也很乐意 采用。花旗会员银行会先把电子型式的账单送到打印机，印成纸账单后塞进信封， 用邮递走1500英哩到加州再拆开，上面印着烂广告的笨信封就丢到垃圾桶，而书 面账单则扫描进计算机，然后放到应该一开始就直接送达的网站上。虽然这种兼 容模式很夸张，不过最后终究会消失，因为PayMyBills.com毕竟不像微软，它可 以让客户真的来用这个系统，所以不久就可以去找愚蠢的花旗会员银行谈：「嘿， 我这里有93,400个你们的客户。为什么不直接和我们联机然后每个月省下93,400 美元呢？」于是突然间PayMyBills.com就变得非常赚钱，而同时微软还在努力 要签下第二个电子化公用事业(或许某家乔治亚州事业会是个好的备案。)

不能认清鸡生蛋蛋生鸡问题的公司可以视为想煮涕大海（boil the ocean)的公司: 他们的商业计划书需要有九千三百万人配合他们疯狂的想法后才能实现。我看过 最无可救药的笨点子叫ActiveNames。他们的猪头点子是让全世界所有人都在电 邮软件上装个附加软件，它会在该公司的中央服务器用人名搜寻取得真实的电邮 地址。这样就不必告诉大家说你的电邮地址是ke「mit@sesame-st「eet.com，只 要说你的ActiveName是「spolsky」就可以了。另外别人想电邮给你时，也得安 装这套特别的软件。哔！答错了！我根本懒得开多冷数落这个点子为什么行不通。

结论：如果你身在一个有鸡生蛋蛋生鸡问题的市场，最乂/有个向后兼容答案来解 决这个问题，否则就得花很长很长的时间伩如无穷久)才能开始。

还是有很多公司认清面对，并且聪明地解决鸡生蛋蛋生鸡的问题。当Transmeta 公开他们的新CPU时，长久以来第一次有赛Intel的公司终于承认一件事，就是 如果想把CPU卖给很多很多人时，就必须能执行x86的程序。Hitachi、Motorola、 IBM、MIPS、国家半导体还有其他数不清的公司都欺骗自己，认为自己有权利发 明一组新指令集，而Transmeta架构则从一开始就认定，任何打算做一台不能执 行Excel的电脑的商业计划书都是不可行的。

三十八、策略III：让我回去!

当你想让人们由竞争者的产品改用你的产品时，必须了解进A摩涛，而且还要了 解得比你想象中深入，否则大家是不会换过来的，而你只好一直等了。

在某篇之前的策略书中，我写过两种公司间的差异：试图赢得现有竞争的Ben and Je「「y型公司，以及试图在无竞争的新领域「抢地盘」的Amazon.com型公司。 当我90年代初期在微软参与Excel工作时，微软是个标准的Ben and Je「「y型公 司。当时已确立的竞争者Lotus 123几乎独占了电子表格市场。当然有些新用户 买的计算机就附有Excel，不过大体上说，如果微软想卖电子表格，就得让用户 切换过来。

这时最重要的事就是承认它。有些公司连这样都做不到。我上一家公司Juno的管 理阶层就不愿意承认A0L已经取得支配地位。他们会说：「还有几百万人还没上 网。」还说：「每个市场都可以容纳两家厂商，就像时报(Time)和新闻周刊 (Newsweek)，可口可乐与百事可乐等等。」他们唯一没说的就是「我们必须让 大家由A0L换过来。」我不知道他们在怕些什么。或许他们认为自己害怕「唤醒 睡狮」吧。某位Juno的明星程序员（不是我)很有#，肆无忌惮地在公司会议上 问了个简单的问题：「我们为什么不多做点什么把A0L的用户抢过来？」结果他 被拖出去骂了一个小时，还取消了本来答应的升迁。（猜猜看他后来去哪家公 司？）

进入一个有竞争的市场并没有错。事实上即使像eBay那么有全新型态的产品，还 是会有竞争的：车库拍卖！不要太心急。如果你的产品在某方面比较好，就有相 当好的机会让大家换过来。不过你必须策略想的思考，而策略性的思考意味着在 明显的作法外再多想一步。 要让人们改用你的产品，唯一游策游就是濟餘摩涛。想象现在是1991年。主导的 Lotus 123电子表格产品拥有100%的市场。而你是微软Excel的产品经理。问问自 己，什么是切换的障碍？什么事会阻止使用者变成未来Excel的客户？

障碍

1. 他们必须知道excel而且认为它比较好
2. 他们必须购买excel
3. 它们必须购买windows才能执行excel
4. 他们必须购买把现有的123电子表格转到excel
5. 他们必须购买重写不能在excel执行的键盘宏
6. 他们必须购买学习新的使用接口
7. 他们需要内存较多而且较快的计算机

如此类推。在大家成为你的客户之前，把这些障碍想成他们得进行的障碍赛。如 果一开始有一千个跑者，一半的人会被轮胎泮到；没被泮到的有一半爬不过墙； 爬得过去的有一半过绳梯时掉到泥坑里；如此类推到最后可能只有一两个人能克 服所有障碍。只要有八或九道障碍，廣’存乂都会有个理由不买你的东西。

这个计算表示如果想攻下现有的市场，消除切换障碍是必须去做的最重要的事， 因为只要消除一个摩藏可能就能倍销售量。消除两个障碍销量又会再倍增。微 软就由障碍列表着手处理廣’存游项尽：

障碍	解决方案
他们必须知道excel而且认为它比较好	宣传excel， 发送战士用磁盘片， 全国巡回到处示范
他们必须购买excel	原123用户幻城excel时提供特别折扣
它们必须购买windows才能执行excel	制作一个执行时期windows版本，随着excel免费附送
他们必须购买把现有的123电子表格转到excel	让excel能够拂去123电子表格
他们必须购买重写不能在excel执行的键盘宏	让excel能够执行123宏
他们必须购买学习新的使用接口	让excel能够使用lotus的功能键， 让你可以用原本的方式作业
他们需要内存较多而且较快的计算机	等摩尔定律解决计算机运算力的问题

效果非常的好。不断努力消除障碍就慢慢地挖走一些Lotus的市占率。

由旧有的独占转移成新独占时常常可以观察到一件事，就是会有一个神奇的「引 爆点」：某天早上醒来，你的产品就由两成市占率变成八成。这种翻盘通常发生 得.常快（由VisiCalc到123再到Excel，由WordSta「到WordPerfect再到Word，

由Mosaic到Netscape再到 Internet Explore「，由dBase到Access，如此类推）。 发生的原因通常是最后一个进入障碍已经消失，于是突然间大家很自然地就切换 了。

显然的，解决显而易见的进入障碍是很重要的，不过当你自认已经处理了明显的 障碍时，必须继续找出那么不那么明显的障碍。而这就是这个策略微妙之处，因 为不明显的事物也会阻止大家切换。

这里有个例子。今年夏天我大部份时间都待在海滩边的一间屋子里，可是我的账 单还是寄到我在纽约市的公寓，而我又时常出去旅行。有个很不错的网络服务 PayMyBills.com应该能让你日子过得更单纯：你把所存的账单都改寄到他们公司， 他们会把账单都扫描到网络上让你到处都看得到。

PayMyBills每个月约九美元的费用感觉还算合理，我也在考虑是否要用，不过我 以前用Internet的财务服务的运气都不太好，比如Datek做我的财务报表就出了 很多算承错误，害我都不敢相信他们是有执照的。所以我愿意试思PayMyBills 却并不喜欢，我希望有退路可以回到原本的作法。

问题是等我用了 PayMyBills之后如果不喜欢，我得逐一通知每一家可恶的信用卡 公司，再度更改我的地址。这可是一大笔工夫。因为我害份万一要来会太 麻烦，所以没有用他们的服务。之前我称这种作法为「秘密锁定」而且稍微赞美 了一下，不过如果潜在的客户发现了这件事就麻烦了。

这就是进入障碍。不是切换进去有多难，而是切换过来有多难。

而这也提醒我一件事，Excel的引爆点大约发生在Excel 4.0推出的时候。而最大 的原因就是Excel 4.0开始能确实写出Lotus电子表格。

没错，你听到我说了。是写出不是读取。结果阻止大家改用Excel的原因，是因 为合作的其他人还在用Lotus 123。他们不想用一个写出的电子表格只有自己能 读的产品：一个典型的鸡生蛋蛋生鸡问题。如果公司其他人都在用123,只有你 一个人是Excel迷，即使你再喜欢，在能融入123环境之前还是不会换用Excel的。

要攻下一个市场就得消除每一个进入障碍。如果你只忽略了一个会泮到一半潜在 客户的障碍，那么，就定义来说，你就不可能获得五成以上的市占率，而且也永远 不能取代主要的竞争者，然后就被卡在鸡生蛋蛋生鸡问题的困境里。

问题是大部份经理人就像拒绝多想一步的棋手一样，一次只会想到一步棋。这种 人大多会说：「让大家转进来用你的产品当然很重要，不过何必浪费我有限的工 程预算让大家转换汸去嘱？」

这实在是很幼稚的策略。让我想起那些独立的书商，他们说：「我为什么要让大 家在我店里舒适地看书？我要他们来买书！」然后某一天Barnes and Nobles在 店里安排沙发和破F#斤，简直就在求大家到他们店里看书就好而不用买书。于是 这些客户全都到他们店里一坐就是刀个yy、_脏手抓著书看。不过客户找到想买 的书的机率，和他们待在店里的时间是成正比的。于是连在爱荷华市最小间的 Barnes and Nobles超市每分钟都可以赚进几百美兀，而独立书店则关门大吉。 亲爱的，曼哈顿上西部的Shakespeare and Company书店并不是因为Barnes and Nobles的价钱低才关门的，真正原因是Barnes and Nobles店里的人比较多。

这个策略较周详的作法并不是对潜在客户施压。如果某人还不是你游客户，试图 锁定他们并不是个好主意。等你有100%市占率时再来跟我谈锁定吧。在那之前谈 锁定实在是太早了点，如果有客户发现你的意图，你只是在花工夫把他们锁在外头罢了。没有人会想切换到一个以后会让他们失去自由的产品。

让我们举个更新的例子，ISP是个很竞争的市场。几乎没有ISP会提供服务在客户 离开后把电子邮件转寄到其他电邮地址。这是个最糟糕的小气想法，而我很惊 讶竟然没有人发现。如果你是个想让大家切换进来的小ISP，人们会担忧最大的 障碍就是要把新的电邮地址通知朋友。所以他们根本不想试用你的服务。即使 试了暂时也不会把新地址告诉朋友，以免过一阵子就不用了。这表示新的电邮根 本不会收到多少信，也表示他们并不会真的试用，当然也不会知道自己是否比 较喜欢你的服务。结果是双输。.

现在假设某个勇敢的ISP作出下面的承诺：「试试我们吧。如果你用了之后不喜 欢，我们会维持你的电邮信箱，还会免费把你的电子邮件转寄到任何其他ISP。 永远有效，你要换几次ISP都可以，只要通知我们，我们就是你永远的转寄服务。」

当然啰，事业经理人会反对。我们为什么应该让客户容易离开服务呢？因为他们 太短视了。那些人观在是你游客户，试图在他们成为你的客户之蔚锁定他们， 只会把他们锁在夕A妥。不过如果你做了承诺让他们在不高兴时可以容易离开， 突然间就消除了另一个进入障碍。就如前面我们所学到的，即使只消除一个进入 障碍都会对转换有重大的作用，随着时间流转，等你敲掉最后一个进入障碍时， 大家就会开始涌入，而生活也暂时变得美好。直到某人对你做一模一样的事。

三十九、策略IV：大件与80/20神话

微软的旗舰电子表格软件Excel的5.0版在1993年问世。这套软件很巨大，需要整 整15 MB的硬盘空间。当时我们还保有自己的第一台20mb的PC硬盘(大概是1985年）， 所以15mb当然感觉很大。

当Excel 2000上市时，它需要恐怖的146mb…，几乎增加了十倍！那些该死的微软 程序员真混，对吧？

错。

我敢打赌你认为我要写一篇网络上到处都是的那种哀叹「肿胀软件」的文章，会 抱怨这些软件实在太肥了，我觉得好难过。edl in和vi都很苗条，所以它们比Word 和Emacs好太多了等等。

哈哈哈！我骗到你了！我并不是要写那种文章，因为那种文章写的并不是事实。 就1993年的硬盘价格来看，微软Excel 5.0占了约36美元的硬盘空间。

而就2000年的硬盘价格来看，微软Excel 2000只占用了约1.03美元的硬盘空间。 (这些数字的依据这里的硬盘价格数据，并依据通货膨胀调整过。）

说真的，看起来Excel其实还变小了！ 肿胀软件究竟是什么？ Jargon File讽刺地把这个字定义为「提供最小功能的软 件，却需要大到不成比例的磁盘空间及内存。特别用于应用程序及操作系统系统 升级。这在Windows/NT界很常见，也因此才产生这个词。」

我猜这些家伙只不过是痛恨Windows罢了。自从Windows 386(1989)出现虚拟内存 开始，我已经十年以上没有用尽内存了。另外硬盘空间已经掉到每MB只要0.0071 美元，而且还像由树上跳下来想学飞的羊一样直线下降中。

或许Linus Akerlund可以解释这件事。他在自己的网站上写道：「使用这些肿胀 程序的最大缺点，就是即使只想完成很微细的事情，还是得加载这个很大的程序。 它会吃光你所有的内存。你并未有效率地使用你的系统。而且整个系统感觉起来 比实际上要慢，而这其实是完全不必要的。」

哦，它会吃光你所有的内存。我了解了。呃，事实上是不会的。远自1997年的 Windows 1.0开始，操作系统就只会在用到内存页时才会加载。如果有个15mb的 执行档而你只用到占了2mb内存页的程序，就只会由磁盘读2mb到RAM。事实上如果 你有更新版的Windows，操作系统还会自动把这些内存页在硬盘上重新整理成连 续排列，这样下次执行时会更快。

我也不认为任何人能够否认，对现今能力过剩且价格超低的计算机来说，加载大 程序的速度还是比五年前的系统加载小程序更快。究竟还有什么问题呢？

RADownes提供了一个线索。他似乎花了好几个小时拆解一个微软小工具，显然 因为这个工具需要1 MB(在他写那篇文章时约当3.15美分的硬盘空间）而生气，他 认为那个程序应该可以小到原本的5%。好笑的地方在于他拆解的程序叫做 RegClean。这个你或许从未听过的程序会搜遍整个Windows登录数据库，找出并 删除掉没有用的东西。一定是有些妄想强迫倾向，才会在意要清除登录数据库里 无用的东西。因此我开始怀疑对肿胀软件的不满比较接近精神健康问题而非软 件问题。

事实上肿胀软件的存在有很多好理由。首先如果程序员不用考虑程序的大小，就 可以更早完成。而这表示你可以得到更多功能，使用这些功能会让你的生活过 得更好，就算不用也没有什么伤害。如果你的软件厂商在出货前停下来，花两个 月把程序缩小到一半大小，你所得到的净利益是微乎其微的。或许(只是或许） 你的硬盘通常都是满的，而又有一首MP3想下载。不过新版软件多等两个月的损 失是看得到的，而软件公司放弃两个月销量的损失更是可怕。

很多软件开发者都被古老的「80/20」定律瞒骗了。这个定律似乎很合理：80% 的人只会用20%的功能。所以你说服自己，只需要实作20%的功能，还是能卖出80% 的数量。

很不幸的是那20%是会变的。每个人都会用不同的功能集。过去十年间我可能已 经听过数十家公司决定不管其他公司的经验，尝试发行只实作20%功能的「精简 版」字处理器。这种故事从PC问世起就有了。通常这些公司会拿程序给某个记者 评论，记者就会用这个新文书处理器来写评论文章，记者都需要知道精确的字 数，找「字数」功能时却找不到，因为这是属于没人在用的80%功能。于是记者 最后就写了一篇报导，试图声称精简版是好的，软件太肥是不好的，而这烂东西 不能算字数所以我没法子用。如果这种事每发生一次我就能拿到一块钱，我会非 常快乐的。

在开始营销你的「精简」产品时，你会告诉大家：「嘿，这东西很小，只占1MB。」 大家通常都会很高兴，然后就会问你有没有他们要的重要功能，如果没有的话还 是不会买你的产品。

结论：如果你的策略是「80/20」，是很难卖出软件的。这就是现实。这个策略 和软件业本身的一样老，不过就是不能奏效；真奇怪为什么有这么多快完蛋公司 的执行长认为会行得通。

Jamie Zawinski讨论改变世界的最初版Netscape时说得最贴切：「虽然这若是 事实的话会很省事，不过Mozil la [Netscape 1 .0]并不是因为有一堆无用废物才 变大的。Mozil la是因为你的需要很多才会大，而你的需要多是因为Internet很 大。外头有很多瘦小的web浏览器，顺带一提的是他们几乎都没什么用。我们写 Mozilla的目标可不是想做一颗完美无暇的闪亮珠宝。。」

四十、策略V：公开源代码的经济因素

我读大学时修了两门经济课程：总体经济和个体经济。总体经济满是「低失业率 导致通货膨胀」等现实中不太能成立的理论，不过个体经济则是酷又有用。里头 都是讨论供需间关系而且真正可用的概念。举例来说，如果你的对手降价而你不 跟进，你的产品需求就会减少。

我想在今天的文章指出，其中一个概念如何能解释某些熟悉的计算机公司。我注 意到开放源码软件有件事很有意思，就是大多数砸大钱发展开放源码软件的公司， 并不是突然爱上言论自由放弃资本主义，而是认为这是个好的商业策略。

市场上所有的产品都有替代激逼和互存激逼。替代物品是首选商品太贵时会改买 的另一种东西。鸡肉就是牛肉的代替物品。如果你是鸡农而牛肉的价格上升，大 家都要买更多鸡肉而你的销量就增加。

互补物品是通常会和其他产品一起购买的产品。车和汽油是互补物品。计算机硬 件是典型的计算机操作系统互补物品。褓姆则是高级餐厅用餐的互补物。当小镇 的五星级餐厅提供两人同行一人免费的情人节特惠方案时，当地的褓姆也会忙得 不得了（事实上连九岁的小朋友要出动了。）

其他事都一样，当商品的价格下降时互补物品的需求就会增加。

让我重复一遍以免你打瞌睡没看到这个重点。当商品的价格下降时互补物品的需 求就会增加。举例来说，如果到迈阿密的机票降价，当地旅馆的住房需求就会 上升，因为会有更多人去迈阿密就会需要房间住。当计算机变便宜就会有更多人 买，由于计算机要有操作系统，于是操作系统的需求就增加，而操作系统的售价 也水涨船高。

这时候大家常常会出来打岔，他们会说：「啊哈！可是Linux是免费的！」好吧。 首先经济学家谈到价格时是指整体价格，里面包括安装时间、人员再教育、现有 流程转换等无形的事物。这所有的东西我们称之为「整体持有成本」。

再来这些支持者会因为免费啤酒的说法，试图相信自己不受经济定律限制，因为 他们有个很棒的零，而任何数字乘以都没有了。这里有个例子。Slashdot包Linux

开发者Moshe Ba「未来的Linux核心是否能与现有设备驱动器兼容，他回答不需要: 「专属软件每行除错好的程序代码收费50到200美元。这样的价格不适用于开放 源码软件。」Moshe继续声称，每次Linux核心改版让现有驱动程序失效是正常 的，因为重写现有驱动程序的成本是零。这实在是错得离谱。他等于是声称维持 核心向后相容所花的小工夫，等价于重写所有驱动程序的大量程序作业，因为两 者都会乘以各自的「成本」，而他们相信这个成本都是零。这是一个似是而非的 谬论。修改现有设备驱动器需要开发人员投入数千甚至数百万小时，一定会产 生某些费用。而且在这些工作完成之前，Linux因为不能支持现有硬件，在市场上等于又再变回残废。把这些「零成本」的工夫拿来改良Gnome或支持新硬件不 是更好吗？

不管是专属软件还是开放源码软件，除错绝对不会是免费的。即使没有实际付钱， 还是有机会成本和时间成本。投入开放源码计划的志愿性程序人力是有限的， 所有开放源码计划都在争取同一群有限的程序设计资源，只有最具吸引力的计划 才能拥有超过需要的志愿开发者。总而言之，我不认同这些试图证明啤酒免费 这种荒唐经济的人，因为我认为他们只会得到「零除」的错误。

开放源码并不能免于重力或经济法则。由Eazel,A「sDigita的事就可以知道，这 家公司以前以VA Linux和很多其他企图而闻名。不过有些在开放源码世界中很少 人真正了解的事还在继续进行：很多有责任尽量提升股东价值的大型上市公司， 投入很多资金支持开放源码软件(通常是负担大型程序团队的开发费用）。而这 正可以用互补物的原理来解释。

再说一遍：当商品的价格下降时互补物品的需求就会增加。通常公司的策略利益 是让互补商品的价格愈低愈好。理论上能维持住的最低价格是「普及价格 (commodity price)」，也就是会有很多竞争者提供相同商品时的价格。所以： 聪明的公司试图让产品的互补物普及化(commoditize)。 如果你能做到这一点，产品的需求就会上升而你就可以卖贵一点然后赚更多钱。

当IBM设计PC架构时是用现成的零件而非订制零件，而且他们很小心地编写零件 间的接口文件IBM-PC技术参考手册(革命性的作法）。为什么要这样做呢？因为这 样子其他制造商就可以进来一起做。只要你遵循接口，就可以用在PC里。IBM的 目标是要普及附加产品，而附加产品正是PC市场的互补物品。他们做得相当成功。 在短时间内就涌现大量的公司提供内存卡、硬盘、显示适配器、打印机等商品。 便宜的附加产品意味更大的PC需求。

当微软把操作系统PC-D0S授权给IBM时，很小心的没有卖出独占授权。于是微软 就可以把相同的东西，授权给Compaq和其他几百家利用IBM自己的文件合法复制 IBM PC的OEM厂商。微软的目标是要让PC市场普及化。不久PC本身也成为日用品 了，价格愈来愈低，功能持续提升，而且也很难有获利的空间。而低价当然就会 提升需求。PC需求增加表示其互补物MS-DOS的需求也增加了。.没有例外的，产 品需求愈大赚到的钱就愈多。难怪比尔盖茨买得起瑞典而你不行。

今年微软想再玩一次：他们的新游戏主机XBox也是用普及的PC硬件而非订制零件。 他们的理论(这本书里有解释)普及的硬件每年都会愈来愈便宜，所以XBox可以 顺势降价。不幸的是他们弄巧成拙：显然PC硬件已经被压到普及价格，所以XBox

的制作成本下降并不如微软预期快。微软XBox另一个策略是使用DirectX，利用 这个绘图链接库写的程序能在所有的视讯芯片上执行。这里的目标是想普及化视 讯芯片并使它降价，就能卖出更多真正能藉以获利的游戏。那么为什么视讯芯 片厂商不想办法去普及化游戏呢？因为浪难做到。虽然最后一战你a/o)这个游戏 疯狂大卖，可是并没存真正的替代品。你并不会跑去电影院想看最新的星际大战， 却又认为改看伍迪艾伦的电影也可以满足你。这些都是好电影，不过绝对不能彼 此替代。好啦，你想当游戏发行商还是视讯芯片厂商呢？

让你的互补物品普及化。

要解释为何这么多商业公司对开放源码投入很大的贡献，了解这个策略有非常大 的帮助。让我们来看看。

头条：IBM投入数百万美元发展开放源码软件。

神话：他们会这样做，是因为Lou Gerstne「读了GNU Manifesto后决定自己其实 不喜欢资本主义。

现实：IBM是因为自己已成为IT顾问公司才会这样做。IT顾问是企业软件的互补 物品，所以IBM必须让企业软件普及化，而最好的方法就是支持开放源码软件。 看吧，他们的顾问部门靠这个策略大获全胜。

头条：Netscape公开其Web浏览器的源码。

神话：他们这样做是为了取得某些纽西兰网咖客贡献的免费程序代码。

现实：他们是要让web浏览器普及化。

及一方:始这就已经是Netscape的策略了。看看Netscape早期第一份公开新闻：这 个浏览器是「免费软件」。Netscape发放浏览器是为了要让服务器来赚钱。浏览 器和服务器是典型的互补产品。浏览器愈便宜，卖出的服务器就会愈多。这在1994 年有最真实的呈现。当MCI进门把那么多钱倒在他们身上时，Netscape是真的吓 到了，他们了解到浏览器本身也可以赚钱。这时当初的商业计划书没有写到的。

Netscape把Mozilla以开放源码方式发行，是因为他们看到降低浏览器开发成本 的机会。这样就能以较低的成本获得普及化的利益。

稍后A0L/时代华纳并购了 Netscape。服务器软件应该是浏览器普及化的受益者， 可是卖得不好就被抛弃了。问题来了：为什么A0L/时代华纳要继续投入开放源码 呢？

A0L/时代华纳是个娱乐公司。娱乐公司是各种型式娱乐传递平台（包括web浏览器) 的互补。这个巨人集团的策略利益在于让娱乐传递(也就是web浏览器)普及到没 有人能因而收费。

Internet Explore「免费的这个事实有点动摇我的论点。微软也想让web浏览器普 及化，这样他们才能贩卖桌上型及服务器的操作系统。他们想再进一步提供一组 元件，让大家都能快速拼凑出一个web浏览器。Neoplanet和A0L还有Juno就用这 些组件建立他们自己的web浏览器。既然IE免费，那么Netscape还有什么动机让 浏览器「更便宜」呢？答案是Netscape要抢先机，他们必须防止微软的web浏览 器完全独大，即使是免费的web浏览器也不行。因为理论上如果微软独占的话就 有机会以别的方式（比如Windows增加的价格)增加网络浏览的成本。

(显然Barksdale当家时，Netscape确实不了解状况，这也使我的论点更加不稳。要解释Netscape的行为，比较可能的说法是高层管理不了解技术，所以只好被开 发人员牵着鼻子走，而不懂经济的开发人员巧合地想出能配合他们策略的机制。 不过还是让我们当他们是真的聪明吧。）

头条：Transmeta雇用Linus，花钱请他研究Linux。

神话：他们纯粹是为了宣传。否则有谁听说过Transmeta呀？

现实：Transmeta是家CPU公司，而CPU天生的互补产品就是操作系统。Transmeta 想要让操作系统普及化。 头条：Sun和HP付钱请Ximian研究Gnome。

神话：Sun和HP支持免费软件是因为他们喜欢市集(Bazaars)而非大教堂 (Cathedrals)。

现实：Sun和HP都是硬件公司，他们做的是盒子。为了要在桌上型市场赚钱，他 们需要窗口式系统(桌面计算机的互补物)成为普及化的产品。那么为什么不把 付给Ximian的钱拿去发展一套专属的窗口式系统呢？他们以前试过(Sun有NeWS 而HP有New Wave)不过这些公司骨子里都是硬件公司，软件技术非常的粗糙， 而且他们需要让窗口式系统成为蘑价普及淄，而不是要付钱取得的专属优势。所 以他们雇用Ximian的优秀人员来做，理由和Sun买下Star Off ice然后开放源码一 样：让软件普及化然后由硬件赚更多钱。

头条：Sun开发出Java这套全新的”Bytecode”系统，写一次程序可以到处执行。

Bytecode的想法并不新，程序员一直在尝试让自己的程序尽可能在更多的机器上 执行(这就是让你的互补物普及化的方法）。多年前微软就拥有自己的p-code编 译程序和可移植的窗口层，让Excel能在麦金塔、Windows、OS/2等系统以及 Motorola、Intel、Alpha、MIPS和PowerPC等芯片上执行。Quark也有一个中间 层可以在Windows上执行麦金塔的程序。C程序语言最佳的描述方式就是一个独 立于硬件的汇编语言。对软件开发人员来说这并不是新点子。

如果任何地方都执行执行自己软件，硬件就更像是普及化产品了。而硬件价格下 降会驱动更大的软件需求(而且让客户由硬件省下更多钱去买如今会更贵的软 件。）

Sun对WORA的狂热可说是奇怪游，因为他们是家硬件公司，而硬件公司最不应该 激游事就是让硬件普及化。

真是令人太惊讶了！

Sun是计算机界有名的怪公司。他们无法看透自己对微软激烈的恐惧及痛恨，所 以其策略都是基于激愤而非自身利益。Sun的两大策略是分别是（a)推广并发展 免费软件(Star Off ice、Linux、Apache、Gnome等）让软件普及化，以及（b)推广 Java及其bytecodeWORA架构和让硬件普及化。好吧，来问Sun—个问题：当音乐 结束时你想坐在哪里？你在硬件或软件上都没有独有的优势，这样只能接受普及 化价格。这种收入只能勉强维持墨西哥的廉价工厂，可养不起你在硅谷舒适的 办公室。

「约耳等一下，」贾里德说：「Sun是想Transmeta像那样让操作系统普及化，并 不是要普及化硬件。」或许吧，不过事实上Java bytecode也让硬件普及化了， 因而会为他们带来相当重大的附带伤害。

这所有的例子里都有一件很重要的事，软件是很容易让硬件普及化的（只要写个 小小的硬件抽象层，比像Windows NT的HAL就只有很小一段程序），不过硬件想让 软件普及化却是难上加难。就如StarOffice营销团队所学到的教训，软件是不能 互换的。即使价格为零，由微软Office切换过来还是需要成本。除非切换成本 变成零，否则桌面办公室软件不可能真的普及化。两套软件间即使差异极其微小， 还是会造成切换时的痛苦。虽然Mozilla的确拥有我所有想要的功能，而且单单 是挡广告的功能就让我会爱用它，可是我实在太习惯按Alt+D跳到网址列了。你 能怎么办？去告我吗？只是一个小小的差异就失去普及化的状态。可是我可以 由IBM的计算机抽出硬盘，直接塞进Dell的计算机里，打开电源整个系统还是完 全正常的启动并运作，就像没换计算机似的。

Creo的执行长Amos M i che l son告诉我，他公司里的所有员工都被要求去听一间叫 「经济思考」的课。真是好主意。因为即使个体经济学的基本观念，对了解现在 发生的重要变化都很有帮助。

四十一、墨菲法则肆掠的礼拜

2003年一月廿五的消息

第一章：存放我们CVS储存库(我们所有的程序）的Linux服务器挂了。没啥大不了 的，我们有自动的远程映对备份(用rd/st)。花了几小时压缩远程数据再传回来。 结果发现我们忘记用让rd/st移除已删除档案的选项，所以备份并不完美：所有 已删除的档案都还包含在内，只能手动删除。

把这事解决之后，我决定把整个程序目录树重新取出，与我原本在用的目录树相 比较，戠当作最后的确认。可是我的笔记本电脑磁盘驱动器空间不足，该是升 级的时候了。我订了一台60 GB的笔记本电脑用硬盘，还有一个PCMCIA/硬盘连接 器好把旧硬盘备份到新硬盘。这个动作做了约六个小时，在完成50%时失败了， 还要我「执行磁盘扫描」。花了几小时扫描硬盘后重来一次。六个多小时后到 50%时又失败了。这次展来游硬盘带着我的一生挂了。我花了几个小时东试西试， 把硬盘接到不同计算机，试尽各种方法，结论是它真的挂了。

好法，这也没什么了不起，我们每天都会备份(NetBackup Pro)。我把新的60 GB 硬盘装进笔记本电脑，格式化后装好Windows XP Pro。然后要NetBackup Pro把 机器回复到挂掉之前的状态。我损失了一天工作时间，不过反正我这一天也做不 了什么事。那天的电子邮件也没了，所以如果你这星期有寄电邮来而我没反应， 重寄一次吧。

NetBackup Pro跑了几个小时，我就回家让它在晚上继续做。结果早上整个系统 都挂了而且连开机都不行。我猜原因一定是我把XP Pro系统回复到W i n2K映像档。 于是我就再做一次，这次先安装Win 2K (硬盘格式化：一小时；安装Win 2K — 小时；再安装NetBackup Pro Client)，接下来再开始回复。五个小时后才做了 一半，所以我就回家了。

第二天系统开不太起来，会出现蓝色当机画面，不过用安全模式试了半小时后终 于能动了。看起来所有东西都回来了，不过有几个我要Windows加密(用EFS)的 档案不能用而且原因不明，应该是公用密钥和认证的缘故。我猜加密过的档案回 复后就会读不出来。我还不知道要怎么解决。如果你知道要怎么解决我会永远感激你的。[1/26:我想尽办法弄了几个小时终于解决了。]

教训：硬盘挂掉后衍生一串问题然后浪费好几个工作日，这已经不是第一次发生 了。注意我有个很不错的备份策略，所有东西每天都会备份到别处。事实上我 相信这已经是第三次因硬盘坏掉引发一串事故而浪费好几天了。结论：备份还不 够好，现在起我要用RAID映对备份。以后硬盘再挂掉，我要能用15分钟换台新 硬盘进去，回到出事前的状态继续作业。新方针：Fog Creek所有非笔电的计算 机都要有RAID映对备份。

第二章。你注意到之前我们的web服务器当了吗？星期五大约中午本地的Verizon 交换器失火，我们的电话和Internet联机全断了。Verizon花了几个小时回复电 话线，不过T1比较麻烦。我们的T1是跟SAVVIS买的，他们则是找MCI(现在叫 WorldCom)来营运地区性的回路。

而Worldcom当然也不是自己实际去蒙运任何线 路，他们才不会弄骯自己的手呢，他们只是向Verizon购买当地的线路。

所以由周五中午一直到周六半夜，Michael和我轮流上场，大约每小时打电话到 Savvis关心一次。我们对Savvis施压，他们偶而就会去催Worldcom，而Worldcom 则认定全都是某种SQL Server DD0S攻击搞的鬼，所以不太理Savvis。而Savvis 也没告诉我们Worldcom不理他们。于是我们再逼Savvis，而勉#7又逼Worldcom， 大概搞了三回Worldcom才同意打电话给Verizon，然后Verizon找了一个技师来把 问题修好。说实在的，这有点像在推弹費。就像上次Savvis让我们的T1线断了一 天，技术问题其实没什么，如果不用跟这么多白痴公司打交道，几分钟就能找到 问题并修好。

教训：当你向某家公司购买服务，而该公司其实只是把服务外包，这样多了一层 是很难有好的客户服务的。如果外包了两次就几乎不可能有什么服务了。我非 常讨厌鼓励独占的地区电信公司，只有一件事会比直接和地区电信公司打交道更 糟，就是和另一家只能和地区电信公司来往的官僚公司沟通。我们以后的办公 室用的是Verizon DSL，真是谢天谢地。

顺便一提，如果Dell把我们该死的服务器准时送到，你们其实完全不会发现这次 的事件。我们应该在一个月前就换到很好的Pee「1高备援安全托管设施上运行。 看看我上一篇大骂。我有提到我发烧了吗？每次事情出错时我就会生病。

第三章。已经发生上千次了，Fog Creek所在的赤褐色建筑四楼的暖气又没了。 暖气是用热水流过墙壁里面产生的。在墙里怎么可能结冻呢？噢，原来上星期火 炉坏掉了，因为当初装火炉的白痴(大概是无照的)装了 25呎长的水f烟囱，不但 妨碍通风还害一个房客住院，火炉也因此熄灭了几十次。最后暖气公司的人承认 有可能加装一个导风管强迫烟囱通风，不过是在热水管冻住之后再施工。水管本 身的隔热当然也没做好，这是另一个无能的纽约建筑业者的杰作。不过如果火炉 正常燃烧是不会有影响的。

教训：差的系统可能表现完全正常，不过等相邻的系统故障时就会出事。有过敏 背痛的人可能好几个有都没事，不过一患干草热打喷嚏就会背痛到不行。这种 事在系统管理中到处都有。可以利用这种机会把所存你题一次修好。帮你所存游 PC都加装RAID龙互定期备份，不要使用EFS，要买很大的硬盘(这样就永远不必停 下工作来升级），另外要再次确认rdist的命令行选项。安装导风管龙互把水管隔 热做好。把你重要的服务器移去安全的代管设施龙互在办公室改用Verizon的T1 服务。

四十二、微软公司是如何败北API之战的

这阵子你应该听过某种理论：「微软要完蛋了。只要Linux在桌上型市场有些进 展，而web应用程序又取代桌上型系统的应用程序，这个强盛帝国就会崩溃了。」

虽然Linux的确是微软的心头大患，不过至少要预言这个Redmond公司灭亡还言之 过早。微软银行里的现金多得不得了，而且仍是非常的赚钱，还要很久很久才 可能倒。微软可以胡搞个十年才会开始有点危险，而且你永远不会知道他 们可能在最后一刻变身成刨冰公司。所以别这么快就看衰他们。90年代初期大 家都认为IBM彻底完蛋了，因为大型主机（mainframe)已经成为历史！那时候 Robert X. Cringely预言主机时代会在2000年一月一日结束，届时所有用COBOL 写的程序都会出问题。由于原始码都早已遗失（据称），所以没有人会去修正这件程序，大家都会针对主从架构的平台把这些程序全部重写过。

好吧，猜猜结果如何。大型主机仍然与我们同在，2000年一月一日什么事都没发 生，而IBM则是变身成一家巨大的老牌技术顾问公司，同时恰巧也在制造便宜塑 料电话。所以只由少数几个数据就推断出微软要完蛋的理论，实在是有点夸大了。

不过大多数人并未注意到一个较不为人知的现象：微软在战略上最重要的宝物 Windows API要输了。Windows与Office的利润丰厚，几乎贡献微软所有的收入， 还养活大量不赚钱或赚很少的产品线。这些赚钱产品的特权和微软的垄断势力 都是以Windows API为基石。不过它对开发人员不再那么有吸引力了。生金蛋的 鹅还没死，不过已经得了还没人注意到的绝症。

既然我已经说了，请容我为前一段的大言不惭和炫耀说声抱歉。我想我看起来开 始像那些商业小报的评论主笔，喋喋不休地谈论Windows API这个微软的策略资 产。我打算在这里用几页的篇幅，来解释我真正的意思并阐明我的论点。在我解 释清楚之前请不要直接下任何结论。这会是篇很长的文章，我得解释什么是 Windows API;我也得说明它为什么是微软最重要的策略式资产，还要解释它会 怎么输掉以及输掉这件事长期的含意。另外既然是在谈大趋势，难免也得夸大其 词和泛泛空谈啰。

开发者、开发者、开发者、开发者

还记得操作系统的定义吗？它负责管理一台计算机的资源让应用程序能执行。大 家其实并不太在意操作系统；比较重视那些依赖操作系统的应用程序。像是文书 处理器、实时通讯、电子邮件、应付帐款管理、有Paris Hilton照片的网站等等。 操作系统本身并没什么用。大家会买操作系统是因为上面可以执行有用的应用程 序。因此最有用的操作系统就是拥有最有用应用程序的操作系统。

由这里可以合理推出一个结论，如果你想要卖操作系统，最重要的就是让软件开 发者愿意替你的操作系统写软件。所以Steve Ballme「才会跳上舞台大喊「开发 者、开发者、开发者、开发者。」这对微软实在是太重要了。微软没有公然发放 Windows开发工具，唯一的理由就

是怕不小心砍断竞争开发工具商的生路（嗯， 是指还活着的那些）。因为有各种开发工具可以选择，会让他们的平台更能吸引 开发人员。不过他们其实真的邀要发放开放工具。你可以透过他们的Empower ISV 深耕计划，以大约375美元买到五套完整的MSDN Universal (也就是「除模拟飞行外的所有微软产品」）。免费的.NET runtime附有命令行版本的.NET语言编译 程序...也是免费的。现在C++编译程序也免费了。微软尽各种方法鼓励开发者支 持.NET平台，只是为了不要害死Borland这些公司才收敛一点。

为什么苹果和Sun不能卖计算机

好吧，这样说当然是有点蠢。苹果和Sun当然可以卖计算机，但不是在最赚钱的 企业桌面计算机和家庭计算机两个市场。苹果的占有率低到只有个位数，而Sun 也只有自己公司的人在用。（请理解我这里谈的是大趋势，所以当我用「没人」 等说法其实是指「少于一千万人」。请如此类推。）

为什么呢？因为苹果和Sun的计算机都不能执行Windows的程序，就算可以也是要 用某种昂贵的仿真模式勉强执行。记住，大家是为了要执行应用程序才买计算机 的，而Windows上好的桌面应用程序比麦金塔多太多了，所以麦金塔的用户实在 很难当。

附栏「API」是什么东西？

如果你正在写一支程序（假设是个字处理器），你想显示选单或是写档案时得呼 叫操作系统替你完成，这时会用到一组每个操作系统都不相同的函数。这些函数 就叫做API，也就是操作系统(如Windows)提供给应用程序开发者（如写字处理器 或电子表格等等的程序员）的接口。它是一组数量上千，复杂而讲究的函数和子 程序，可以让程序员指示操作系统做些有趣的事（如显示选单和读写档案）、奇 怪的事（如把指定的日期用塞尔维亚语表示），以及非常复杂的事（比如在窗口 里显不一个网页）。如果你的程序使用了 Windows的API，就不能在提供不同API 呼叫的Linux上执行。有时候API做的事是差不多的。Windows软件不能在Linux上 执行有一个重要的原因。因为想让Windows程序在Linux上执行，就得重新实作整 个Windows API。这包括数千个复杂的函数，规模几乎和实作Windows本身一样大， 其中有些东西微软用了数千个人年才做出来。如果你犯了个小错误或是漏了某个 应用程序需要的函数，那个应用程序就会当掉。

而这正是Windows API对微软是极重要资产的原因。

(我知道，我知道，这时候占全世界计算机人口2.3%的麦金塔用户，正要打开电 子邮件程序写信我说他们多么喜爱麦金塔。再次声明，我在谈大趋势和一般化， 所以别浪费你的时间。我知道你爱你的麦金塔，也知道它可以执行所有你需要的 东西。我爱你，你是个好人，不过你只是全部的2.3%，所以这篇文章不关你事。）

微软内的两股力量

微软内部有两股相对的力量，我称之为（虽然有点讽刺意味）Raymond Chen阵营 和MSDN杂志阵营。

Raymond Chen是微软Windows团队的开发人员，从1992年起就在那里了。他的网 志"The Old New Thing"里有很多详细的技术性文章，解释Windows里某些东西为 什么会是现在的样子，甚至很蠢的东西其实也有着很好的理由。

看Raymond的网志时令人印象最深刻的是这些年来Windows团队为了维持向后相 容，投入难以置信努力的故事：

由客户观点来看看这个情境。你买了程序X和Y还有Z，后来升级到Windows XP。现在计算机会乱 当机，而且程序Z还不能动。你会告诉朋友：「不要升级到Windows XP。它会乱当机而且跟程序Z 不兼容。」你会去查看看是不是程序X造成当机吗？或是去查出其实程序Z用了未公开的窗口讯息 所以不会动吗？当然不会。你只会把整盒Windows XP拿回去退费。（程序X和Y和Z都是几个月前 买的。30天内退货的时限已经超过，你也只能退Windows XP了。）

我最初是由热卖游戏SimCity的某位作者听到这些事的。他说他的程序里有只大 虫，会在释放内存后马上又用到，这是个大问题，在DOS上恰巧能动,不过在 Windows就会出事，因为释放的内存立刻就会被其他程序拿去用。Windows团队的 测试人员测遍各种常见的应用程序，要确定是否都能正常执行，可是SimCity 一直当机。他们把这个状况回报给Windows开发人员，开发人员就反组译SimCity 用除错器逐行追查找出问题，然后加上特别的程序检查代码检查SimCity是否正在执 行，如果有的话就把内存配置程序切成特殊模式，让内存在释放后还可以使用。

这并不是特例。Windows测试团队非常巨大，而他们最重要的责任之一就是要保 证不管装了什么程序，每个人都要能安然无事地升级操作系统，而且即使这些程 序做了坏事或呼叫未公开函数，或是依赖在Windows n有但n+1版修好的问题，都 必须能继续执行。事实上如果你在registry登录数据库的AppCompatibility区到 处看看，就会看到一大堆要特别处理的应用程序，Windows会仿真各种旧问题和 古怪的行为让这些程序能继续执行。Raymond Chen写道：「有人指控微软在0S 升级时恶意地妨害应用程序时我会觉得很生气。如果有任何程序不能在Windows 95执行，我会视为个人的失败。我有很多个晚上没睡去修正别人的程序，只是为 了让它们能在Windows 95执行。」

很多开发人员和工程师都不同意这种作法。他们认为如果应用程序做了坏事或依 赖某些未公开的行为，应该让它们在OS升级后直接挂掉。苹果公司的麦金塔OS 开发人员一直都在这个阵营。这也是很少麦金塔早期的程序还能动的原因。举例 来说，很多开发人员为了加速自己的麦金塔应用程序，会把中断跳跃表的指标 复制出来直接呼叫，而不按正常方法使用处理器的中断功能。虽然苹果的官方麦 金塔程序设计圣经/ns/de Macintosh里有段技术注记写着「你不可以这样做」， 这些人还是照样做了，程序会动而且跑得更快…结果等下一版操作系统出来这些 程序就完全不能动了。如果写这些应用程序的公司因而没了生意（大多数的确如此），好吧兄弟，只能怪你们运气太差了。

对照之下，由于Raymond Chen阵营的努力，我1983年在最早期IBM PC上写的DOS 程序还能正常执行。我知道这当然不只是Raymond—个人，这是整个核心Windows API团体的工作稽神，不过Raymond用他出色的网站The Old New Thing把它公布 出来，所以我才用他的名字。

这是其中一个阵营。另一个阵营我称之为MSDN杂志阵营，是以某一本开发人员杂 志来命名，这本杂志充满了让人兴奋的文章，都在教你用各种在自己软件里结 合微软产品的神秘方法来害死自己。MSDN杂志阵营总是试图说服你用新而复杂的外部技术，比如COM+、MSMQ、MSDE、Microsoft Office、Internet Explorer 及其组件、MSXML、DirectX (请用最新版）、Windows Media Playe「、以及 Sharepoint… Sharepoint!这个没夜人有的东西名符其实的有一大堆壮观的夕A 斑关联，当你要把应用程序发行给付钱的客户时，每个关联都会变成大麻烦，而且没有办法弄好。这种事的技术性名称叫DLL地狱。在我这里能动，为什么在那里不会动了？

Raymond Chen阵营相信，让开发人员写一次程序能到处（好吧，在所有的Wi ndows 上）执行，可以让他们更容易做事。而MSDN杂志阵营则认为要提供一些功能很 强的程序给开发人员使用，才能让他们更轻松，前提是开发人员愿意承受难以置 信的复杂部署和安装麻烦，更别提巨大的学习曲线了。Raymond阵营想的是强化 (consolidation)。请不要让事情变得更糟，只要让我们原有的东西能遂’续动就 好了。MSDN杂志阵营则是得一直生产出大量的技术，却没有人能跟得上。

下面是这件事要紧的原因。

微软失去向后兼容的信仰

在微软内部，MSDN杂志阵营已经赢了这场战役。

第一个大胜利是让Visual Basic.NET不必向后与VB 6.0相容。这是记忆中第一次 买了微软产品的升级版后，无法安静而完美的汇入旧魏据（也就是你用VB6写的 程序）。这也是第一次微软的升级版不尊重用户用该产品之前版本所做的成果。

而且天似手还没有塌下来，至少微软内部没出事。VB6开发人员极力反对，不过 反正他们也正在消失中，因为这些人大多都是企业开发人员，反正正在转移到web 开发。真正的长期伤害就被隐藏起来了。

MSDN杂志阵营挟着这次大胜接管一切。突然间改东西变得理所当然了。IIS 6.0 用了不同的线程模型，让某些旧应用程序不能动。我很震惊地发现用Windows Server 2003的客户不能执行FogBugz。然后.NET 1.1不能完全向后兼容于1.0。

而现在秘密终于揭露，0S团队也心领神会，决定不再为Windows API增加新功能 而是完全取代掉。我们被告知不要再用Win32了，现在要开始准备迎接WinFX:下 一代的Windows API。全部都不一样了。现在依据的是用受控代码(managed code) 的.NET、XAML、Avalon。是的，我得承认远远优于Win32。不过这并不是升级， 而是对过去的破坏。

Windows开发的复杂困扰了外界的开发人员，他们被微软磨个平台打败了，现在 已经在发展web平台了。在dotcom兴旺初期建立了 Yahoo! Store的Paul Graham 很有力地总结：「新创公司现在有更多的理由去写Web-based软件，因为桌面软 件写起来已经不那么好玩了。现在想写桌面软件就要照微软的规矩，呼叫他们的 API还要应付他们多虫的OS。另外如果你真的写出什么热门的东西，可能会发现 自己只是在替微软做市场研究。」

微软已经大到有太多的开发人员，这些人太沈溺于增加收益，因此他们突然决定 把每洋事彻底重做过并不:是太了不起的计划。该死的，我们还可以做两次啊。 旧的微软（Raymond Chen的微软）可能会把Avalon (新的绘图系统）这种东西实 作成一系列的DLL，不但能在任何版本的Windows上执行，还可以和需要用到的应 用程序包在一起。并没有任何技术上的理由不这样做，不过微软必须找个借口 让你买Longhorn。他们想弄出大量的改变，就像Windows取代DOS时那么多的变 化。问题是Longhorn并没有比Windows XP先进太多；根本不到Windows超越DOS 的那种程度。或许它的吸引力并不足让人像对Windows那样，为它买全新的计算 机和应用程序。好吧，或许它有这个能耐，微软一定得让它有，不过就我目前 所看到的实在没什么说服力。微软很多东西都赌错了。举例来说，WinFS的宣传 是以关系数据库方式制作文件系统，以达成搜寻功能的一种作法，却忽略了事实 上要达成搜寻功能的真实作法就是要达成搜寻功康Ct/?e rea/ way to ma/ce searc^/ng wor/c /s by ma/c/ng searc^/ng wor/c)。不要让我替所有档案输入提 示数据然后用查询语言去搜寻。只要帮我个忙去搜寻该死游硬盘，决遽姻找至夕我 打的字符串，随便你用全文检索或其他W73年就有的技术。

自动排档获得最后胜利

不想把我想错了…我认为.NET是个很伟大的开发环境，我也相信搭配XAML的 Avalon比Windows旧GUI应用程序的写法先进许多。.NET最大的优势就是拥有自动 化的内存管理。

我们很多人都认为1990年代最大的战争就是程序化程序设计与面向对象程序设 计间的战争，而且我们认为面向对象程序设计能让程序员的生产力大幅提升， 我当时也是其中之一，而有些人到现在也还是这么认为。不过结果我们错了，面 向对象程序员设计是很方便的好东西，不过并不能像它承诺地大幅提升生产力。 真正让程序员大幅提升生产力的，其实是那些会替你自动管理内存的程序语言。 它可以是参照计数（reference counting)或垃圾收集（garbage collection); 可以是Java、Lisp、Visual Basic (连1.0版也算）、Smalltalk或是多种脚本语 言其中之一。如果你的程序语言能让你抓一块内存来用，又不用考虑用完后要如 何释放，你用的就是会管理内存的程序语言，那么你的效率会远远超过那些使 用得明确管理内存的语言的程序员。当你听到某些人夸耀他们的程序语言生产力 有多好时，他们的生产力可能大多是自动化内存管理所贡献的，只是他们弄错 原因而已。

> 附栏 > > 为什么自动化内存管理能大幅提升生产力？ 1)因为你可以写f(g(x))却不用担 心要如何释放g的传回值，这表示你的函数可以回传很复杂数据型态，而这样的函 数能让你以更高阶层的抽象想法来作业。2)因为你不必花任何时间写程序代码去 释放内存或追查内存漏洞(memory leak)。3)因为你不再需要小心安排函数的离 开点以确保内存都有释放干净。

赛车迷可能会因而写信骂我，不过就我的经验而言，在正常驾驶时好的自排只有 在一种状况下会不如手排。软件开发也是类似的：几乎在所有的状况下，自动化 内存管理都比手动内存管理更好，而且能让程序员生产力提升许多。

在Windows初期要开发桌面应用程序，微软提供两种作法，可以写C程序直接呼叫 Windows API并自行管理自己的内存，或者用Visual Basic并把内存交给它管理。 这也是我个人过去13年来用得最多的两个开发环境，用到熟得不得了，而我的经 验是Visual Basic的生产力好赛常多。我时常会写褙局游程序，用C++呼叫 Windows API写一次，用Visual Basic也写一次，C++通常要花三到四倍的工作时 间。为什么呢？答案是内存管理。要了解原因最简单的方法，就是去看任何会传 回字符串的Windows API函数文件。仔细看看有多少篇幅在讨论该字符串的内存 由谁配置，或是如何协商需要的内存数量。通常你得呼叫函数两次，第一次告诉它你配置了 0个字节，函数会传回「配置内存不足」的讯息，顺便还告诉你需要配置多少内存。这还是简单的情形，如果运气不好呼叫到的函数要传回字符审游 审/f或是整个长度不定的结构就更麻烦了。不管如何，像开档案写入字符串然后 关闭档案之类简单的操作，直接呼叫Windows API都需要一整页的程序代码。在 Visual Basic类似的操作只要三行。

所以你有了这两种程序设计的世界。每个人都断定受控代码的世界远比未受控代 码世界更优越。Visual Basic是有史以来（恐怕现在还是）卖得最好的程序语言 产品，而且开发人员在发展Windows程序时会优先选它而非C或C++。不过产品名 称里有”Basic”这个事实却让硬派程序员回避，虽然它其实是个相当现代的程序 语言，有很多面向对象功能而残留的脏东西也极少（行号和LET叙述早就不见了）。 VB的另一个问题是部署时需要附上VB runtime。这对透过调制解调器散播的共享 件是个大问题，而更糟的是VB runtime会让其他程序员发现你的应用程序是用 (可耻的！）Visual Basic开发的。

一个Runtime全部搞定

接下来又出现了.NET。这是个宏大的计划，一个要把所有脏污一次清干净的超级 伟大一统计画。当然它会有内存管理，也有Visual Basic不过已经是种新语言。 精神基本上还是与原Visual Basic—样，不过改用大括号和分号等类似C的语法。 而最好的一点是这个新的Visual Basic/C合体叫做Visual C#，所以再也不用对 别人说你是一个”Basic”程序员了。所有那些可怕的Windows函数，加上它们那些 尾巴和挂入功能（hook)还有向后兼容问题与不可能弄懂的字符串回传机制全 部一扫而空，取而代之的是个单一干净只有一种字符串的面向对象接口。一个 Runtime全部搞定。真是漂亮。就技术面而言他们也的确成功了。.NET是个伟大 的程序设计环境，可以管理你的内存并提供一组丰富完整且一致的操作系统接口， 另外还有一组丰富而超完整又优雅的对象库提供各种基本的操作。

不过，大家还是没有真正大量使用.NET。

噢，当然某些人是有啦。

不过建立一个完全兹激的程序设计环境来一统Visual Basic和Windows API程序 设计的混乱，而且这个新环境并不是用一种或二种语言，而是有三种程序语言（或 许是四种吗？），这种作法实在有点像用压倒性的音量大喊「闭嘴！」让两个 小孩停止吵架。这种作法只有在电视上行得通。在真实世界里，如果你对两个大 声吵架的人喊「闭嘴！」，结果只会变成三个人在吵架。

(顺便一提，网志聚合器格式是个神秘而充满政治味的世界，有在注意的读者可 以看到那里面发生的事情是一样的。RSS已经变得支离破碎了，原因是有数个不 多的版本，规格不正确又有很多政治斗争。有人试图建立叫Atom的另一#游式来 消弭混乱，结果只是在RSS的众多版本外再加一个Atom而已，规格还是不正确而 政治斗争依旧很多。当你创造第三方案想藉以一统两股对抗的力量时，最后只会 变成三股敌对的力量。你并不会一统什么也没有真的修正什么。）

于是我们现在并没有出现.NET的一统和单纯，反而是拥有六重的复杂，每个人都 试图在这团乱中找出所用的开发策略，以及是否有本钱把应用程序移植到.NET。

不管微软的市场信息有多么一致（「只用.NET就对了？把我们当白痴啊！」）， 他们大部份客户还是在用C、C++、Visual Basic 6.0以及原本的ASP，更别提其 他那些来自别的公司的各种开发工具了。而在用.NET的都是在用ASP.NET来开发 web应用程序，只会在Windows服务器上执行并不憲要W/'ndows游客户端系统。这 正是个关键点，后面写到web时会深入探讨。

噢，等一下，还有其他东西要出来！

现在微软有太多的开发人员在做事，彻底重做整个Windows API还不够看，他们 必须重做两次。去年的PDC中他们预先发表了下一个重大的操作系统改版，这个 代号Aon^orn的系统除了其他东西之外，将会有一组代码为Aw/on的全新使用 接口 API。这组API是运用现代计算机快速显示硬件及实时3D成像能力重新建立的。 如果你现在正在开发Windows GUI应用程序，而且是用微软「官方」最新最伟大 的Windows程序设计环境WinForms的话，为了支持Longhorn和Avalon两年内就得 重来一遍了。这说明了为什么WinForms完全的难产。希望你在这上面还没有投资 太多。Jon Udell找到一份标题为「如何在Windows Forms和Avalon间选择？」微 软的投影片，里头问道：「我们为什么要在Windows Forms和Avalon间选择一个呢？」真是个好问题，而且还是个他找不到好答案的问题。

所以你有了 Windows API，有了 VB，现在还有.NET，虽然有多种程序语言可选，

不过不要跟任何一种牵涉太深，因为我们正在制作Avalon而你知道它只能在最 新的微软操作系统上执行，而这个系统要等很久很久才会出现。就我个人来说 还没空很深入的学习.NET，而我们也没有把Fog Creek的两套产品由传统的ASP及 Visual Basic 6.0移植到.NET，因为投资完全不?会存浓解。以我来看这只是边开 火边移动的掩护行动。微软会很乐意看到我们停止为我们的问题追踪软件和内 容管理软件增加新功能，然后浪费几个月把它们移植到别的程序开发环境上，这 个动作不会对任何一个客户有好处，因此也不会让我们多卖出一套软件。这

对 微软非常好，因为他们有内容管理软件也有问题追踪软件，因此他们最高兴我浪 费时间空转追逐流行，然后再浪费一两年做Avalon的版本，而同时他们却在自 己的相同产品上一直加新功能。完全丑■磁。

有正职的开发人员不会有时间追得上来自Redmond的所有新开发工具，因为微软 有太多该死的员工在做开发工具!

这不是1990年代

微软是在1980和1990年代长大的，当时个人计算机的成长非常的快速，每年新卖 出的计算机都超过原有的计算机数量。这也表示如果你做的产品只能给新计算机 用，即使没有人改思你的产品，一两年内还是可以攻下全世界的市场。这也是 Word和Excel能如此彻底地取代WordPerfect和Lotus的原因：微软只要等下一波 硬体升级，把Windows和Word以及Excel卖给更新桌面计算机企业就好了（有些 还是第一次买计算机）。所以微软在很多方面从来不需要学习让现有的客户由 第N版产品转换到N+1版。当人们拿到新计算机时，会很乐意在新计算机上搭配微 软所有的新东西，不过升级的意愿就低很多了。当PC产业如野火般成长时这并不打紧。不过现在这个世界的PC已经饱和了，而且大部份的PC都用得很好。谢谢你，

微软突然了解到最新版的东西没那么好推了。他们想把Windows 98完全结束，结 果还在使用的人实在太多，于是他们只好承诺会对那个老爷系统再多支持几年。

这些勇敢的新策略（.NET、Longhorn、Avalon之类的玩意）试图建立一组，新API 把大家锁住。问题是大家都还在用1998时买来还很好用的计算机，这种策略是 不太行不通的。即使Longhorn照计划在2006推出（我根本不相信），大概还得多 等几年，拥有的人数才会多到能考虑作为开发平台。开发者们才不会听从微软 那些多重人格失调的软件开发建议呢！

进入Web

我不知道自己怎么能写这么久还没提到Web。每个开发人员在计划新软件应用程 序时都要做一个选择，他们可以做成web版本，也可以做一个在PC上执行的”rich client”应用程序。基本的优缺点很单纯：Web应用程序比较容易部署，而rich client应用程序反应较快所以使用界面比较有趣。

Web应用程序比较容易部署是因为不需要安装。Web应用程序的安装等于只是在网 址列输入一个URL。今天我要安装Google的新电邮程式，只要按Alt+D、gmail， 再按Ct「l+Ente「就好了。兼容性问题还有与其他软件相冲的问题都少太多了。产 品所有的用户都在用相同的版本，所以你永远不必支持各种旧版本。你可以用 任何喜欢的开发环境，因为只要装在自己的服务器上执行就好了。在这个星/发上 几乎每台还可以用的计算机，自然而然都能用你的应用程序。而你客户的数据也 很理所当然能用于这星球上任何一台还可以的计算机上。

不过这必须付出使用接口的平顺作为代价。下面是几个web应用程序做不好的例子：

1. 创造一个快速绘图的程序
2. 写一个能标示红色波浪底线的实时拼写检查
3. 在使用者按到浏览器的关闭盒时，警告使用者将会遗失其工作成果
4. 不必连到服务器来回沟通一遍，就能依据使用者的变更来更新小部份的显 示，
5. 建立一个不需鼠标的快捷键盘驱动接口
6. 让大家在没有连上Internet时继续作业

这些都不算大问题，其中有些很快就会被聪明的Javascript开发人员解决。有两 个新的web应用程序Gmail以及Oddpost (都是电邮程序）做得相当不错，避开或 完全解决了部份的问题。另外使用者似乎也不在意UI小问题或web接口的缓慢。 不知道为什么，不管我如何努力宣扬rich client会，呃，比较丰富，我认识的 人几乎全都对web式的电子邮件软件十分满意。

所以Web使用接口已经有80分了，就算不靠新的web浏览器还是有机会达成95分。 这对大多数人来说已经够好了，当然对开发人员来说也够了，而且他们也已经实 际把几乎所有重要的新应用程序都写成web应用程序。

这表示微软的API突然间已经不那么重要了。Web#不需要W/ndows。

微软并不是没注意到这件事发生。他们当然知道，所以他们在局势浮现时就猛踩 煞车。他们不再在未来计划里承诺像HTA和DHTML这类的新技术。Internet Explore「团队似乎也消失了；他们有好几年似乎完全没有动作。微软不可能容许 DHTML变得比现在更好，这对他们的核心事业rich client来说实在太危险了。微 软最近的重大思想基因（memo)是：「微软把公司放未来齢在rich c/ient上。」 这句话会遍布Longhorn相关的每页简报上。来自Avalon团队的Joe Beda说道：「Avalon (大体上还有Longhorn)是微软的基石。这句话表不我们相信你桌面的 威力，能够完成各种了不起的东西，而且是不可或许的。我们正在持续投入桌面， 我们认为这会是个好地方，而且我们希望自己能启动一波振奋…」

问题是现在已经太迟了。

我本身对这有一点点难过

我本身对这真的有一点点难过。对我来说Web是很不错，不过Web应用程序的使用 接口既烂又慢也不一致，就日常使用来说实在是倒退一大步。我爱我的rich c l ient应用程序，如果日常使用的应用程序（Visual Stud io、CityDesk、Outlook、Corel PhotoPaint、QuickBooks)得改用web版本我会疯掉。不过这正是开发人 员正在进行的事。没有人（再提一次，我的意思是「少于一千万人」）想再用 Windows API开发程序了。创投业也因为害怕微软的竞争，不会再投资Windows 应用程序了。而大部份使用者似乎并不像我一样，那么在意不方便的Web使用介 面。

以下是关键所在：我注意到（而且也和求才业的朋友确认）纽约市这里会C++和 COM程序设计的Windows API程序员年收入约十三万美元，而一般用可控代码语言 (Java、PHP、Perl、甚至ASP.NET)的普通web程序员年收入是八万美元。这可 是很大的差别，而当我和从事微软顾问服务的朋友谈到这事情时，他们承认微 软已经失去整个世代的开发人员了。要花十三万

雇一个有COM经验的人，是因为 过去约八年来没有人自找麻烦去学COM程序设计，所以你得找个很资深的人来 (通常都已经晋身管理阶层)，说服他们再做个普通程序员去处理(上帝救救我) marshalling、 monikers、apartment thread i ng、aggregate、tearoff，还 W其 他一百万件基本上只有Don Dox会的东西。事实上连Don Box都无法忍受再回来看 这些东西。

虽然我很不想这样说，不过大量的开发人员早就移到web而且拒绝再回来。大部 份的.NET开发人员都是用ASP.NET针对微软的web伺服器进行开发。ASP.NET很出 色。我从事web开发已经十年，而它真的是比其他工具先进一个世代。不过ASP.NET 是服务器技术，所以客户端可以用任意的桌面系统。何况ASP.NET还能在Linux 上用Mono执行得相当好呢。

这些东西没有一个对微软有利，对微软得力于API权势的利益也一样。新的API 是HTML，而能让HTML唱歌的人将会是应用程序开发市场的新赢家。

第四部分 对.NET稍多的评说

四十三、微软精神失常了

微软最新宣告的产品Microsoft .NET被财星杂志之流誉为巨大的「变革」，不过 实际上只是个泡沫软件，而我认为这证明了 Redmond有些事错得很严重。

利用泡沫软件可以承诺各式各样你根本不能卖的功能和产品，因为实际上这些东 西你都没有。不过.NET比泡沫软件更糟。以微软那种漠不关心的高傲态度来看， 他们根本连泡/本衣身都懒得提供。

仔细看看这份白皮书就会发现，在整场闹剧中.NET只是一团恐惧和不确定以及 怀疑的疑云。里头没有内容。你可以尽量去挖，可是整份白皮书什么都没说。于 是你抓得愈用力，从指缝间漏掉的东西愈多。

我并不是说.NET里面没有新东西。我是说里头什么都没存。

看看这些叙述：

所有人都相信Web将会演进，不过要真正落实到开发人员和企业以及消费者，还 需要一个基础的新愿景。微软的目标就是提供这个愿景以及实现愿景的科技[摘 录自「微软.NET:实现新世代」，2000六月]。

这个又如何：

微软.NET愿景意味把权力落实到消费者、企业、软件开发人员以及整个业界。它 释放Internet的所有潜能。它是照你想的方式运转的的web。 [Ibid]

这里头究竟怎么回事？整份白皮书里找不到一个能在软件产品里实际执行的想 法。微软并不是提供功能列表，而是列出一串如下的无形「好处」：

网站成为可互动、交易并能运用彼此数据的灵活服务。[Ibid]

这就是这个振奋.NET架构的「功能」之一。它实在是太空泛而且高高在上，椒本 不康表示在何东西。不过这个事实似乎没有对任何人构成困扰。再来看看这个：

微软.NET提供了寻找服务和人并与其互动的可能性。[Ibid]

噢，真好笑！ Altavista出来已经五牟，而Larry Page和Sergei Brin发明一个先进许多的搜索引擎也有两年了，微软还在假装没有方法搜寻Internet，而他们要 替我们解决这个问题。整份文件都是这种东西。

这里有两件事。微软有些很伟大的思想家。当伟大的思想家考虑问题时会开始看 出模式。他们看到人们要传送字处理档案给别人，又看到人们要传送电子表格给 别人，然后就会发现里面有一个通用的模式：传送档案。这已经是一层的抽象。 然后他们会再上一层：人们传送档

案，不过网页浏览器也会「传送」网页要求啊。 这些全都是传送的动作，所以我们聪明的思想家就发明了一种更新更广义的新抽 象概念：「讯息传送(messaging)」，不过现在这件事变得愈来愈神秘，再也没 人真的懂他们在说些什么。

而当你一直往上把事情弄得太抽象，就会像上太空一样没有氧气。有时候这些聪明的思想家就是停不下来，然后就创造出这些荒唐又无所不包的高层次宇宙景像， 这些东西什么都好，就是完全没有实际的意义。

而这似乎就是这里所发生的事。

下一世代的Windows桌面平台Windows.NET支持生产力、创造力、管理、娱乐以及 更多更多，并设计让用户掌控他们的数字生活。[Ibid]

这种说词实在太抽象了，根本无法批评。谁会不要一个支持生产力的操作系统呢? 真是伟大的功能！请给我一套这种有生产力功能的新神奇操作系统吧！问题是 微软要怎么达成呢？软件界20年来的生产力改善都是渐进累积的。难道他们发现 一种能让他们的操作系统更具生产力的新化合物吗？我不认为他们有这种东西。 我认为他们在吹牛。都是恐惧、不确定、怀疑，以及泡沫软件。

让人害怕的是他们是认真游。

我在微软工作过三年，了解这家公司。我了解写这份文件的那种人。几乎可以确 定比尔盖茨在里头占了很重要的角色；这也是他放弃CEO位置的原因，这样才能 参与这件事。我不认为微软写这份文件只是为了拥有一些泡沫软件。他们可都是 超级聪明。

我真的认为他们认真地以为自己正在创造未来，而且也自认知道怎么做到。他们 审视由Hotmail到SQL Serve「每一个微软的产品，尝试把这些产品套入一个奔放 的新愿景中。不过问题是里面并没有人真的发明什么惊天动地的东西。这并没什 么好惊讶的，不是因为微软笨，他们并不笨，只是惊天地动的新发明太罕见了 而微软的聪明人数有限。全世界只有一个人发明Napste「，而他并不在微软做事。 微软自暴自弃地想相信自己能量产革新，不过即使在这个Ｉnternet的寒武纪大 爆发期，每年还是只会出现几个真正革命性的想法，而这些点子之一出现在比尔 盖茨小世界和Redmond武士团的机会实在是太小了。何况一个相当聪明的程序员 可能在微软里写Windows NT的显示驱动程序，他即使有很伟大的点子可能也没有 人会听倒。考虑到这种状况后微软想出绝世好点子的机会就更小了。

这份白皮书里唯一歳看清的具体内容，就是软件应该是一种经由Ｉnternet取得的 订阅服务，而不是由光盘安装的某种东西。

对一个客户来说，本来由光盘安装的字处理器改由Ｉnternet取得可能只有少淨的 益处，不，可能根本没有好处。这些做并没有实际解决任何客户的问题。可以 透过Ｉnternet修正问题？真伟大。不过我已经可以做到了。七年前起我就已经在 网络上下载微软产品的修正文件了，

而且现在都是自动进行的。取得新版本？ 如果新版本的唯一功用就是更容易取得新版本，这有什么意义！最新的三版Word 几乎没有增任何一个新功能，真要说有的话就是做了些奇怪的事，让图片的定 位比较「容易」，结果我根本没办法把图移到想要的位置。

事实是过去1991年时微软注意到升级使他们的收益成长。由于很难强逼大家升级， 于是他们努力了几近十年，试图让客户同意用订阅方式购买软体。不过这是行 不通的，因为客户并不想要这种模式，这只是满足微软替一万名程序员安排未来 十年工作的需要罢了。我们都知道，他们想出一个大家都要的文书处理功能已 经是很久之前的事了，没有好点子的话这些程序员要做什么呢？

「愿景空话」的好处

有个老笑话：有个男的去看精神科医生。医生要他看一张鸟的照片并问他：「这 会让你联想到什么？」男人回答：「性。」医生拿一张树的照片再问：「很好， 那么这会让你联想到什么？」男人回答「性」。换火车照片，还是「性」。房子， 也是「性」

「我的天啊！」医生说：「你满脑子都是性！」

「我满脑子都是性！？」男人说：「是你一直拿色情照片给我看耶！」

你看，像.NET白皮书这种空乏文件的好处在于它就像某种罗夏克测验(Rorschach Test)。大家都带着预设立炒读文件，由于这份文件如此的含糊不清，所以大 家就认为微软在覆述他们的想法。UserLand software的总裁Dave Wine「对软件 有很多有趣而创新的想法。当他读到微软.NET相关的文章时，他认为微软终于认 清楚他讲了两年的想法。Dave，你太瞧得起他们了。他们跟你比起来实在太无 知了。他们是在玩心灵电话热线和报纸占星术的把戏：讲些无意义模糊的泛泛之 词，让你落入陷阱误认为他们能感应你的想法。「今天行星连成一直线，表示你 向自己的目标前进了一大步。」其中的差异在于Dave有真实明确的想法可以转换 成真的软件，而微软还停在六年前喊着Cairo会提供「指尖上的信息」那个奇幻 世界，而当初那个愿景却是由Ｉnternet而非Cairo所达成的。

所以只能希望这一切无意义的废话真能让某人的创意继续前进(就像UserLand那 样），然后导致某些真正的创新。不过这些创新比较可能出现在微软外面而不是 内部。

后记：「可是等一下，」你说：「我有NET SDK啊！」

这篇文章一出来，就有很多人写来告诉我他们有.NET SDK!这并不是泡沫！这是 「真的」！

好吧，嗯哼，是。里面有什么呢？有SOAP这个基本上是Dave Wine「依据XmlRpc 创出的技术，后者我本人大概两年前就用在Juno的登入程序里了。微软玩这种游 戏有点晚了。还有个叫C#的程序语言，这只是微软的一种表示方式，表示他们如 果不能控制Java，就要回家玩他

们自己的玩具。另外还有新版的ADO和ASP以及其 他东西…都是好东西，不过只是渐进式的改进。财星杂志注意了，这里头并没 有革命性的内容。即使微软的营销人员没有拼命推销，反正我们还是会拿到这些 东西的，而且没有人会假装这是某种水平线那头的计算机天堂。

这就是微软的作法：每个产品都有一个产品团队，每个团队每隔一两年就要推出 新版本的软件。就是这样而已。先生太太们，我们这里有的只是一个纯粹的营 销团队，他们到处看看所有快要出来的新版本，决定需要一个「主题」让微软看 起来像个革命性的大创新者，于是命令大家把下一版的东西叫__.NET”。当你在 微软这样的地方工作，没有什么事比营销人员接管一切更让人沮丧了：看看微软 内部人员对我这篇文章的回应。

四十四、我们的.NET对策

以下是我目前对Fog Creek内渐进移到.NET开发工具的想法。

现状：CityDesk大部份是用Visual Basic 6.0写的，小部份则是用Visual C++6.0。 FogBUGZ大多是用VBScript for ASP写，小部份是用C++。几乎我们所有的内部工 具和我们的web呈现(Fog Shop，讨论区等等)都是用VBScript for ASP。

为什么要要操心移完全到.NET嘱？简单说是因为.NET是目前为止最出色而且生产 力最高的开发环境。ASP.NET真的让web应用程序写起来容易得不可思议；过去 这几天我已经以出奇的高速写出几个内部用的应用程序。各种苦工杂事(如表格 验证及错误报告)在用ASP写web应用程序时会耗去75%的时间，在ASP.NET却变成 轻松的小事。ASP.NET的生产力远远超越ASP，其间的差别就像Java和C一样。了 不起。

C#有很多Java的优点，还有一些小的改进如自动的boxing。虽然我们过去用ASP 和VB6时总是尽量写出相当面向对象的程序，不过移到C#还是不错的。

最后是.NET所附的类别链接库真的很好。事实上由数据存取到web开发再到GUI开 发，每祥东西都重新设计过，所以由上到下到难以置信的协调。举例来说，当 你检视原本的Win32 API，就会惊讶由函数呼叫取得字符串竟然有那么多种方法。 每隔两年他们就会改变主意认为另一种方法比较好。.NET把这些问题全部清掉了。 我很高兴现在有一个ASP.NET日历构件(widget)可以用，它会产生由月历选一个 日期的HTML码，而且可以放心那个东西传回的日期类别(我相信是 System.DateTime)和SQL Serve「类别所用的日期类别一样。你大概不会相信，以 前我们浪费了多少时间去针对SQL叙述调整日期格式，或是把COleDateTimes转成 其他日期时间格式。再来终于出现了！一个到处都可以用的字符串类别！我上 星期才在写一段ATL程序在BSTR、OLECHAR、char *、LPSTR还有其他乱七八糟的 型别间转来转去。真是大解脱啊。

好吧，我承认，.NET违反了绝不要从头重写的规则。微软有两个条件可以这样做。首先他们有全 世界最好的程序语言设计者，过去20年间软件开发的生产力提升，有九成都是这个男人贡献的。 他是Anders Hejlsberg，赐予我们Turbo Pascal (谢谢你！）、Delphi (谢谢你！）、WFC(好尝试！） 和现在的.NET(把球打出场啰！）。其次是他们有本钱投入了无数工程师在这上面做三年，这期间 他们的竞争力或多或少都会被延误到。记住，微软可以做并不表示你也能做。微软创造他们自己 的重力。正常的规则是对他们无效的。

有些人现在会开始写些气愤的信赞扬其他环境，或是质问我为什么不用可以写一 次到处用（窃笑）的Java，或者用Del phi (天才已经离开那里了。.NET'就是Del phi 7.0, 8.0, 9.0)或Lisp或其他东西。他们会说我已经被锁在微软的卡车上了！可 惜我现在没时间来讨论宗教，而且我也觉得讨论这东西很无聊。我才不在意就语 言来说日文是否比英文更好。这根本就无关紧要。让我们继续谈完我们的策略。

第一个问题：我对.NET还没有懂到能写出好的程序代码。在任何开发环境中，要 做某件事时通常都会有很多方法，而我们连第一种方法都还没学会，更别提第二 种方法了。所以我们写出的.NET程序代码质量还没有好的能当产品推出。Bill Vaughn第一本谈ADO的书出来之前，我们连基本SQL查询的最佳作法都不知道呢。 所以我们最优先的事是教育，方法是用.NET来制作往后所有内部用的软件和web程序（基本上就是没有人会付钱的软件）。我们可以把部份的Fog Shop移到.NET， 各种内部的东西一定会用.NET。（今天我用ASP.NET写了一个FogShop优待卷产生 器。写得一团乱不过会动！）

第二个问题：肥大的20 MB CLR (runtime)。8 MB的CityDesk下载文件里有6 MB

来自执行时期链接库和数据存取链接库，已经是够惨了；我们绝不可能期望每个 CityDesk Starter Edition使用者另外再下载20 MB的东西。只能希望一年或两 三年后很多人由其他地方拿到CLR了 (Windows XP没有附真是太惨了）。我们会注 意状况；它以往都会改掉你的user agent;希望现在还是一样，这样我们就能拿 到统计资料。

底线是现在不能把C ityDesk或是FogBUGZ移植到.NET。当CLR普及率到75%时我们 就会移植CityDesk的未来版本。计划如下：

1. 用微软的转换工具移植现有的程序代码和窗体
2. 先用暴力法修正问题，直到能动为止
3. 用C#建立新窗体及类别
4. 以渐进方法，在旧的窗体及类别一定得大改时移植到C#
5. 很多旧窗体和类别只要还正常动作，就会永远维持在VB.NET(使用 丑陋的向后兼容字符串函数等等）

FogBUGZ也需要等CLR在服务器计算机的普及率变得更高；我们必须对我们的客户 进行调查，看看如果FogBUGZ要用CLR时状况会有多糟。

我们有另一个正在开发但还不能公开讨论的产品；这个产品和FogBUGZ共享了很 大量的程序代码(一个我们称之为”Dispatcho”的子集合），所以在我们移植 FogBUGZ 之前都还是用VBScr i pt/ASP。

至于FogBUGZ/D ispatcho/秘密新产品，计划如下：

1. 等到需要CLR也不会出问题的时候，
2. 把现有的「商业逻辑」类别移植到C#
3. 现有的web窗体还是维持用ASP，
4. 用ASP.NET建立新的web窗体。

四十五、请问，我可以使用连接程序吗

原因不明，不过微软最先进又很出色的.NET开发环境少了一个重要的工具…一个 约1950年起在软件开发环境里就很普遍的工具，这个工具是理当所然应该有的， 真奇怪竟然没人注意到.NET真的没有。

少了的工具是什么？就是链接程序啦。链接程序做的事如下，它会把你的程序编 译过的版本，和所有有用到的链接库函数编译过的版本结合起来。然后会把你没 用到的所有链接库函数都拿掉。最后再产生大家可以在自己计算机上执行的单一 可执行二位程序。

.NET用了一个叫”runt ime”的想法来取代链接程序。这是一大团22 MB并会动态链接的程序代码，每个人在用.NET应用程序之前计算机上都得先有这东西才行。

Runtime是个和DLL很像的问题，因为版本1的应用程序在设计上是要配合版本1 的runtime，而等版本2的runtime出来，版本1的应用程序突然间就会原因不明 地不太正常，这时候你麻烦就大了。举例来说我们把runtime由1.0升级到1.1之 后，现在公司的控制面板会把销售数字取成小数点后4位的数字。通常不相容的 状况要比这更糟。

事实上.NET包含了一个叫”manifests”的大型技术系统，这个系统显然很复杂，本来的用意是要确保各个应用程序只会用到正确的runtime，不过我认识的人没 有一个知道要怎么用。

这衍生了一个故事。在Fog Creek的新年除夕宴会上，我们要让主要房间里的很 多计算机屏幕在午夜前倒数。Michael用C#的WinForm写了一只程序，只用了60 秒左右。真是个伟大的开发环境。

我的工作是把countdown.exe拷到三台计算机上执行。听起来很简单。

才怪。在执行档上连双击就看到一个荒谬又神秘的错误讯息，说mscoree.dll还 是什么东西有问题，然后就是没有意义地倾印了我的路径。一点也没提到问题其 实只是没有安装.NET runtime而已。还好我是个程序员，知道这一定就是问题。

你要怎么安装runtime呢？「最简单」的方法是用Windows Update。不过Windows Update—定要我先取得所有重大的更新后才能装runtime。这很合理，对吧？「重 大」更新里面有两个分别是Wi ndows se「v i ce pack和新版的I nternet Explorer， 这两个都得重开机。

就这样，为了要在这几台机器执行这小小的.NET应用程序，我得下载大约70或80 MB(还好我们网络够快)然后重开机三四次。而这还是在一家软件公司里！我知道 这花的确实时间，因为我一开始下载就在大电视屏幕上放映上班一条虫(Office Space)，电影放完时也几乎快装完了。电影每播十分钟左右我就得起来，到每台 计算机前去按那些笨对话盒的确定按钮。

这就我们自己用的程序来说已经够令人沮丧的了。不过再想想我们的产品 CityDesk。我们的使用者几乎每个人都会在购买前下载一份免费试用版。下载文 件的大小约9 MB而且不需要任何其他东西。而这些使用者几乎全部都还没有.NET runtime。

如果我们要求我们的试用者(通常是小型组织或家庭用户），只为了试用我们的应 用程序，却要经历整部电影长的痛苦安装过程，我想我们大概会损失95%的潜在 客户。这些人还不：是客户，只是期望的客户，我实在无法承受只为了一个较好用 的开发环境而放弃95%的期望客户。

「不过，」人们会说：「有runtime的人数总会变多，最后人数够多这个问题就 不见啦。」

我也这么认为，然后我了解到微软每六到十二个月就会推出的runtime，于 是逐渐增加的已安装人数又跳回零。如果我得痛苦挣扎在三种runt i me版本上测 试我的应用程序，只为了多那1.2%有其中一个版本的客户，那我还真该死啊。

我只想把所有用到的东西链接成单一个静态的执行文件，不用先装任何东西就能 执行。我不在意执行档是否会大一点。我要的只是我实际展至如勺函数、bytecode

直译器和小小的执行时期程序。我不需要属于runtime—部份的整个C#编译程序。 我承诺CityDesk不会编译到任何C#原始码。我也不需要全部的22 MB，我要的最 多只有五或六MB吧。

我知道有些公司拥有这种技术，不过没有获得微软许可不能重新发送runtime里 的数据(如byte code直译器）。所以微软醒醒吧，给我们一些1950年代的链接程 序技术吧，让我可以制作单一个执行档在Win 98及后续系统上执行，而且没有其 勉夕A班关联。否则.NET对消费者下载软件来说有致命的缺陷。